THE EXPERT'S VOICE® IN OPEN SOURCE

Foundations of Agile Python Development

Python, agile project methods, and a comprehensive open source tool chain!

Jeff Younker

Apress®

Foundations of Agile Python Development

Jeff Younker

Apress®

Foundations of Agile Python Development

ISBN-13 (pbk): 978-1-59059-981-5

ISBN-10 (pbk): 1-59059-981-0

ISBN-13 (electronic): 978-1-4302-0636-1

ISBN-10 (electronic): 1-4302-0636-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Tom Welsh
Technical Reviewer: Will McGugan
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Susannah Davidson Pfalzer
Copy Editor: Damon Larson
Associate Production Director: Kari Brooks-Copony
Production Editor: Elizabeth Berry
Compositor: Dina Quan
Proofreaders: Nancy Bell, April Eddy
Indexer: John Collin
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com`.

Contents at a Glance

About the Author xiii
About the Technical Reviewer. xv
Acknowledgments xvii
Introduction. xix

CHAPTER 1 What Is Agile Development? 1
CHAPTER 2 The IDE: Eclipsing the Command Line 21
CHAPTER 3 Revision Control: Subverting Your Code 41
CHAPTER 4 Setuptools: Harnessing Your Code 81
CHAPTER 5 A Build for Every Check-In 103
CHAPTER 6 Testing: The Horse and the Cart 139
CHAPTER 7 Test-Driven Development and Impostors 175
CHAPTER 8 Everybody Needs Feedback 233
CHAPTER 9 Databases 263
CHAPTER 10 Web Testing 309
CHAPTER 11 Functional Testing 339

INDEX 369

Contents

About the Author xiii
About the Technical Reviewer. xv
Acknowledgments xvii
Introduction. xix

CHAPTER 1 What Is Agile Development? 1

Why More Methodologies? 1
A Little History 3
Planning and Agile Development 4
What Are Agile Methods?. 4
Pair Programming 5
User Stories 7
The System Metaphor. 8
On-Site Customers 8
Unit Tests 9
Test-Driven Development. 10
Refactoring 11
Simple Design 12
Collective Code Ownership 12
Short Iterations. 13
Continuous Reflection 15
Continuous Integration 16
Documentation. 17
Summary. 18

CHAPTER 2 The IDE: Eclipsing the Command Line 21

Installing Eclipse 23
Installing Plug-Ins 25
Installing and Configuring Pydev. 31
Your First Project 32
Looking Under the Hood. 38
Paying for More Functionality 39
Summary 40

CHAPTER 3 Revision Control: Subverting Your Code 41

Revision Control Phylum. 42
What Subversion Does for You 43
Getting Subverted 44
Working with Your Subverted Code 47
    Examining Files 49
    Adding Files 50
    Copying and Moving Files 51
    Deleting Files 52
    Reverting Changes. 53
    Modifying a File 53
Updating Your Working Copy. 54
Conflicting Changes 55
Subverting Eclipse. 59
    Sharing Your Subverted Project 59
    Importing from Subversion. 60
Working with a Subverted Eclipse 64
    The Team Repository View. 65
    Adding a File. 68
    Committing Changes. 70
    Editing a File. 71
    Reverting Changes. 72
    Resolving Conflicts 73
    Deleting Files 76
    Moving Files 77
    Renaming Files. 77
    Copying Files 78
    Reverting Moves, Renames, and Copies 79
Summary. 79

CHAPTER 4 Setuptools: Harnessing Your Code 81

The Project: A Simple RSS Reader 81
Python Modules 82
The Old Way 83
The New Way: Cooking with Eggs 84
Some Notes About Building Multiple Versions 85
Installing Setuptools 86
Getting Started with Setuptools 87
Building the Project. 88

Installing Executables . . . 91
Dependencies . . . 92
Think Globally, Install Locally . . . 94
    Removing an Existing Package: Undoing Your Hard Work . . . 95
    Installing from the Local Copy . . . 96
Fixing Options with setup.cfg . . . 97
Bootstrapping Setuptools . . . 97
Subverting Subversion: What Shouldn't Be Versioned . . . 98
    The Easy Way with Eclipse . . . 100
Checking in Changes: Not Losing It . . . 100
Working in Development Mode . . . 100
Summary . . . 102

CHAPTER 5 A Build for Every Check-In . . . 103

Buildbot Architecture . . . 104
Installing Buildbot . . . 104
Configuring the Build System . . . 106
Mastering Buildbot . . . 107
Enslaving Buildbot . . . 112
Hooking Up Source Control . . . 116
Using the Source . . . 119
Subversion to Buildbot, Over . . . 121
A Python for Every Builder . . . 122
Finally, a Real Build Succeeds . . . 124
Installing the Build . . . 125
Supporting Python 2.4 Builds . . . 128
Ensuring Local Dependency Processing . . . 132
Keeping Up Appearances . . . 134
Summary . . . 136

CHAPTER 6 Testing: The Horse and the Cart . . . 139

Unit Testing . . . 141
The Problems with Not Unit Testing . . . 142
Pessimism . . . 143
Test-Driven Development . . . 146
Knowing Your Unit Tests . . . 147
unittest and Nose . . . 148
A Simple RSS Reader . . . 149
The First Tests . . . 151

Finding Tests with Nose 159
Skipping Slow Tests 160
Integrating the Tests into the Environment 162
Running Tests After Every Change 163
Running the Complete Test Suite in Development 167
Buildbot with Unit Tests 171
Summary 173

CHAPTER 7 Test-Driven Development and Impostors 175

Moving Beyond Acceptance Tests 175
Renaming 183
Overriding Existing Methods: Monkeypatching 185
Monkeypatching and Imports 186
The Changes Go Live 188
Using Data Files 189
Isolation 190
Rolling Your Own 192
Python Quirks 193
Mocking Libraries 193
Aggregating Two Feeds 194
A Simple pMock Example 195
Implementing with pMock 196
Test: Defining combine_feeds 196
Test: Defining add_single_feed 197
Refactoring: Extracting AggregateFeed 198
Refactoring: Moving add_single_feed 199
Test: Defining create_entry 200
Test: Ensuring That AggregateFeed Creates a FeedEntry Factory 200
Test: Defining add 201
Test: AggregateFeed.entries Is Always Initialized to a Set 201
Test: Defining FeedEntry.from_parsed_feed 202
Test: Defining feed_entry_listing 202
Test: Defining feeds_from_urls 203
Test: AggregateFeed Initializes the FeedParser Factory 203
Test: Defining from_urls 204
Refactoring: Reimplementing from_urls 204
Refactoring: Condensing Some Tests 206

Test: Formatting Feed Entry Listings 207
Test: Defining print_entry_listings 208
Test: FeedWriter Initializes the stdout Attribute 209
Test: Empty AggregateFeeds Generate No Output. 209
Test: Defining is_empty 210
Test: Defining new_main 210
Test: The Application Initializes Dependencies. 211
Refactoring: Making new_main the New main2 212
A Simple PyMock Example 212
Monkeypatching. 214
Saying the Same Thing Differently 214
Implementing with PyMock 215
Test: from_urls and Mocking External Modules. 216
Test: Defining add_single_feed 217
Refactoring: Moving Methods to a New Object 218
Refactoring: Moving add_single_feed 218
Refactoring: Moving from_urls() 219
Test: create_entry() and Mocking Class Constructors. 220
Tests: Defining add and AggregateFeed.__init__ 221
Test: Defining FeedEntry.__init__ 222
Test: Defining listing 222
Test: entry_listings Should Be Sorted 223
Test: Defining print_entry_listings 224
Test: print_entry_listings Should Do Nothing with Empty Feeds 225
Test: is_empty and the Unproven Test 226
Test: new_main, Hooking It All Together. 226
Test: RSReader Initialization 227
Finishing Up: Activating the New Functionality. 227
Other pMock and PyMock Features 228
Raising Exceptions with pMock. 228
Raising Exceptions with PyMock. 228
Playback Counts with pMock. 229
Playback Counts with PyMock. 229
Mocking Attribute Setters with PyMock. 229
Mocking Generators with PyMock 230
Using PyMock with unittest 230
Summary 231

CHAPTER 8 **Everybody Needs Feedback** 233

Measuring Software Quality 235
Measurements 236
Quantitative Measurements: How Much Is That Doggie
in the Window? 237
Code Coverage 237
Complexity Measurements 239
Velocity: When Are We Done? 242
Qualitative Measurements: It's a Shih Tzu! 243
Coding Conventions 244
Welcome Back to Python 246
Never Try to Fix a Social Problem with a Technical Solution 248
Code Reviews 249
Renaming 250
Communication 250
Technological Feedback: Bad Programmer, No Cookie 251
Coercion at the Keyboard 251
When Code Is Submitted 256
Buildbot and Coverage 258
Summary 261

CHAPTER 9 **Databases** 263

A New Religion 263
Blurring the Boundaries 264
Concealing Data Access 265
Object-Relational Mappers 265
The Active Record Pattern 266
The Data Mapper Pattern 266
The Unit of Work Pattern 266
Python ORMs 267
SQLObject 267
SQLAlchemy 283
Building the Database 296
Testing 297
Refactorings 298

Migrations . . . 298
The Instructions . . . 299
Numbering Migrations and Playing Them Back . . . 299
Where to Put the Migration Mechanism . . . 300
DBMigrate: A Migration Mechanism . . . 300
Summary . . . 306

CHAPTER 10 Web Testing . . . 309

Really Simple Primer . . . 309
HTML . . . 310
CSS . . . 311
XML . . . 311
URI and URL . . . 311
HTTP . . . 312
JavaScript . . . 312
Web Servers and Web Applications . . . 312
WSGI . . . 314
Using the write Callback . . . 315
WSGI Middleware . . . 316
Testing Web Applications . . . 316
Graphics and Images . . . 317
Markup . . . 317
Testing JavaScript . . . 320
Using JsUnit . . . 321
Running a Test . . . 322
How It Works . . . 326
Connoisseur of the Undefined . . . 327
Adding a Little More Realism . . . 328
Manipulating the DOM . . . 328
Aggregating Tests . . . 335
Running Tests by URL . . . 336
Summary . . . 337

CHAPTER 11 Functional Testing 339

Running Acceptance Tests 339
PyFit. 340
    Writing Requirements 341
    A Simple PyFit Example 344
    Giving the Acceptance Tests a Home. 346
    Your First FIT. 346
FIT into Buildbot. 353
    Preparing the Slave 353
    Run New Builder, Run! 354
    Making the Reports Available 358
    Getting Regular Builds 366
    What's Left? 367
Summary. 367

INDEX 369

About the Author



JEFF YOUNKER is chief engineer of Data-Pipes (`www.data-pipes.com/`). His educational background carefully avoided computers, but he was drawn in anyway. Most of his misguided adulthood has been spent in large installation systems administration, tool smithing, and release engineering, with a peculiar obsession involving both monitoring and rapid deployment. Over the last several years, he's had the pleasure of working with Python full time. Having escaped Texas nearly a decade ago, he now lives in gray and rainy Northern California. When not suffering monitor-induced radiation burns, Jeff likes to do anything that doesn't involve a roof, unless the roof has been top-roped or covers a machine shop.

About the Technical Reviewer



WILL McGUGAN is a software developer and author currently working in London on a social networking site for games built with Django. See his blog at `www.willmcgugan.com/` for more information on Will's work and open source projects.

Acknowledgments

I'd like to thank Apress and Jason Gilmore for giving me the opportunity to write this book. Thanks to Tom Welsh for the unfaltering criticism that has made it something more than my incoherent ravings. I thank Susannah Davidson Pfalzer for playing midwife to Jason's child after he left the Apress family. This has been a huge undertaking. If I had really understood the magnitude, I might not have started to begin with, but Susannah kept everything on track and graciously coped with the events in my life impinging on the schedule.

Thanks to Will McGugan for the many improvements he made in the code, and for his meticulous attention to detail. Damon Larson did an amazing job turning my technobabble into coherent English. It was wonderful working with Liz Berry in the last few weeks of madness as the disjointed word-processed files turned into something that looks suspiciously like a book.

A whole slew of friends came out of the woodwork as they discovered that I was writing this book, and I was astonished to discover that they actually wanted to help. The last half of this book is much richer for the efforts of Matt Ho, Jacob Hoffman-Andrews, Nancy Huntingford, Rachel McConnell, and Erik Ziko.

In particular, Matt Ho's assistance with Chapter 11 was indispensable. I don't know if I could have finished this book on time without the days we spent in his living room. At a point where I was completely lost, his advice made the path clear. I wish we'd talked much earlier in the process.

My business partner David Birkhead bought time with our customers to give me an opportunity to finish this project, and I owe him a huge debt for that. He's been hugely supportive outside of our business relationship, too. I have to thank our customers Cynthia Walston and David Alban for being so supportive and accommodating during the last month of this process.

I owe a huge debt of gratitude to my father, William James Younker, who passed away somewhere around Chapter 5. No matter how poor my family was, he always saw to it that I had the tools to follow my curiosity. I wish I could show him this book you're reading. At least I'll get to show it to my mother, Hallie Younker, who did an astoundingly good job of bringing me up given the circumstances that enveloped our lives.

I'd like to thank my teachers, too—I had some amazing ones. I'd like to thank Adréa Shaw for giving me such wonderful anecdotes about Baylor College of Medicine. I remember the soft glow of the electron microscope fondly. I'd like to thank David Raikow. He writes more than anyone I know, and his advice kept me moving along when I got stuck. I'd like to give my thanks to Ethyl the Dog, too. She was remarkably good at alleviating writer's block.

I've received support in other ways, as well. Roxanne Williams and Sören Ragsdale lent me technical assistance when necessary. Blair Miller kept me from going insane and from time to time made sure that I stayed fed. Scott Calvert, Gwen Perry, and David Cutler made sure that I wasn't completely isolated from my friends. Kathryn Keslosky's assistance with revisions was immensely helpful.

My dear friends Jaron and Cherry Rothkop made sure that my time in Toledo was bearable. Clare and John Vrakking put up with my mental absence from our first Christmas together in over a decade. I'm sorry this book didn't let me make Passover this year. Finally, I don't know if I could have completed this book without Amy Woodward's love and support. Her calming voice and compassionate words kept me from coming unglued when I became immobilized by the seemingly unending tide of drafts and revisions. She assisted me emotionally and materially, and she believed in me more than I believed in myself at times. I owe her a debt of gratitude that I don't know if I can ever repay. A thousand love songs do her no justice.

Introduction

If you're embarking on a Python development project, then you should buy this book—there's nothing quite like it. I know this because I was looking for it last year, and I couldn't find it. This book introduces the tools you'll need to get started on agile projects in Python, and unlike any other book out there, it shows you how to tie them all together.

Sure, there are many good books on agile development. A lot of them cover the development processes in great detail, and this is a good thing. Agile development is very much about human interactions and the environment surrounding software development, but there is a whole ecology of tooling to make everything work at a practical level.

Agile development eschews extensive up-front specification, and it anticipates that the product will constantly change, but it puts in place rigorous checks to compensate for anticipated change. Testing is an integral part of agile development from the very start, and it is pursued with ferocious rigor. You need software tools to facilitate testing.

Agile projects have very short release cycles, and this has implications for tooling, too. There's no way to have two-week release cycles if it takes you days to integrate changes, days to perform QA, and days to package and deploy the software. This means that agile development puts a high value on build and release automation.

While agile development techniques can be applied to any project, both testing tools and build automation tend to be very language specific. These tools do exist in Python. They're widely available, and by and large they're free, too, but the documentation tends to be . . . um . . . spotty. And while there may be documentation on the individual tools, the documentation telling you how to tie these tools together is usually sparse to nonexistent. This book provides that missing documentation.

Who This Book Is For

This book is written for a person who knows how to program and is already familiar with Python. If you have some Python under your belt and you're thinking of starting a new project, but you don't know how to get started, then this book is for you. If you're an experienced Python programmer and you want to give this agile stuff a whirl, then this book is for you. If you're a release engineer who has been thrown headlong into the world of Python, then this book is for you, too. If you're brand new to programming or don't really know Python, this is not the best book to start with. There are some wonderful books out there that will introduce you to the language, but this isn't one of them.

What's Really in Here?

Each chapter in this book addresses a different aspect of tooling in an agile development environment. These are collected roughly into two parts, with the first focusing on basic tooling, and the second focusing on specific practices. If you're already familiar with Subversion, Setuptools, and Buildbot, then you should have no problem jumping between Chapters 6 through 11. If you're not, then you'll want to look at the earlier chapters first.

Chapter 1: What Is Agile Development?

Chapter 1 provides an overview of the methods that characterize agile development methodologies, with a focus on those not directly related to tooling.

Chapter 2: The IDE: Eclipsing the Command Line

This book uses the command line throughout, but modern IDEs provide many benefits. This chapter introduces you to Python development using Eclipse and the Pydev plug-in.

Chapter 3: Revision Control: Subverting Your Code

A revision control system is part of the core infrastructure for any agile development environment. Subversion is an excellent choice. I show you how to use it from the command line and from Eclipse using the Subversive plug-in.

Chapter 4: Setuptools: Harnessing Your Code

You can't replicate your work for testing purposes without some sort of a framework. In Python, a natural choice is Setuptools, which provides a solid basis for automated builds.

Chapter 5: A Build for Every Check-In

Automated build systems form the core of a continuous integration system. Here I introduce Buildbot, an excellent system that happens to be written in Python. It ensures that the code you check in builds correctly.

Chapter 6: Testing: The Horse and the Cart

Unit testing ensures that your code runs as you expect it to, and it prevents regression (reappearance of old bugs) when you change existing code. I introduce the unit-testing packages unittest and Nose, and I show how to use Nose to run tests from within Eclipse and Setuptools. Finally, I show how to link them into Buildbot.

Chapter 7: Test-Driven Development and Impostors

Test-driven development (TDD) is the practice of writing tests before writing the code they test. Imposters (a.k.a. mock objects) provide a powerful unit-testing technique to isolate units of code. I examine two mock object frameworks, pMock and PyMock, and I work through a sizable example to show how TDD, refactoring, and imposters are used, and how they affect the code that you produce with them.

Chapter 8: Everybody Needs Feedback

Improving your code requires feedback—useful information that sometimes comes from your coworkers, and sometimes from software. Accurate feedback requires standards. This chapter looks at code coverage, complexity measures, and development velocity. It also examines coding standards, how they can be enforced from within Eclipse, and how you can prevent bad code from reaching your repository by using Subversion pre-commit hooks.

Chapter 9: Databases

Databases are very widely used these days, and they pose their own special challenges for agile development. This chapter examines the object-relational mappers SQLObject and SQLAlchemy, and then examines how to version databases using the DBMigrate tool.

Chapter 10: Web Testing

The web is everywhere, and web development has its own set of issues. This chapter examines general approaches to testing web applications, and introduces HTML/XML verification using `ElementTree` and `BeautifulSoup`. It also looks into JavaScript unit testing with JsUnit.

Chapter 11: Functional Testing

This chapter examines functional testing with a particular emphasis on acceptance testing using PyFit. The chapter shows how to use PyFit, and more importantly, how to tie PyFit into Setuptools and Buildbot. (In my view, this alone is worth the price of the book.)

Contacting Me

Finally, please don't hesitate to give me feedback on the book at any time. This is my first book, my writing ability has improved immensely as the book has progressed, and I now have a much better understanding of what I wanted to say than when I started. I'll try to improve any sections that people find lacking and publish them to this book's web page at `http://www.apress.com/book/view/9781590599815`. Additional materials may be available on my blog (`www.theblobshop.com/blog`) under the tag `famip`. I'll present more information in these locations as it becomes available. This pertains but is not limited to notes about anything that I've fouled up, new thoughts, and additional materials that I think you may find useful.

CHAPTER 1

What Is Agile Development?

Agile development is a term given to an entire class of iterative development methodologies. Their unifying characteristic is a focus on short development cycles, on the scale of weeks rather than months. Each development cycle, referred to as an iteration or sprint, produces a working product. This chapter introduces the motivations for the movement to agile software development and surveys the practices that commonly constitute these methodologies.

These practices, in the order to be discussed, are as follows:

- Pair programming
- User stories
- The system metaphor
- On-site customers
- Unit tests
- Test-driven development (TDD)
- Refactoring
- Simple design
- Short iterations
- Collective code ownership
- Continuous reflection
- Continuous integration
- Documentation

Why More Methodologies?

Some projects succeed and some projects fail. This happens regardless of what development methods are used. Development is about much more than simply the techniques that are used. Good development depends upon a strong grounding in reality; not everything can be known before a project starts, and this must be taken into account when planning. Some of these new facts will be minor, and some will be major.

Accommodating major new facts often requires hard choices to be made; making these hard decisions requires sound judgment, and even then, the sound judgments are sometimes wrong. Making these judgments requires guts and integrity; a project leader who is unwilling to stand up and tell the truth potentially sacrifices the development organization's well-being, the product's quality, and possibly the whole organization's long-term viability. This holds true no matter how the project is developed.

These days, the waterfall methodology is a favorite whipping boy. If you're advocating something strongly, then it helps to have something else to demonize, and many agilists have fastened upon the waterfall methodology for that purpose. There's a lot of software out there that has been developed nominally using the waterfall method; whether the engineering staff actually followed the documents is open to question. The waterfall methodology reflects the aspirations of many toward producing better software, and it reflects the best understanding that was available at the time, but there are valid criticisms that have been leveled against it:

It assumes that all change can be predicted and described up front. Software is created to serve a purpose, but if the conditions in the world change, then development needs to change to reflect the new realities. This often happens in the middle of a project. A new disruptive technology is released. New versions of interoperating software are released, altering dependencies and interactions; the new software has features that duplicate functionality being implemented or changes how the existing functionality works. New competitors may come into the market, or the regulatory environment may change. The world is simply too complicated to anticipate all changes up front.

Exploratory tasks that should be part of development are pushed into the design phase. Many judgments about suitability can only be addressed through the creation of prototypes. Many times, determining how something can be designed most effectively requires building a substantial part of it. Why should this effort be wasted?

The waterfall methodology also assumes that documentation can be sufficient to describe a system. There are fields with far more detailed and elaborate documentation systems than are found in software development; mathematics, medicine, and the law are three examples. Nobody in these fields has the hubris to say that documentation alone is sufficient for achieving understanding. Instead, they recognize that a person cannot become an expert without tutelage.

A software specification detailed enough to unambiguously describe the system is specific enough to be translated automatically to software. Such a process simply pushes the effort of coding into design, yet if this is done without feedback from operating models, the design will have errors.

Agile methods emphasize accommodating change, group communication, and iterative design and development. They attempt to cast off excess process. Some of it is just jettisoned; some of it is replaced by other practices. Agile methodologies range from extreme programming (XP), which focuses almost exclusively on the developer and development techniques, to the Dynamic Systems Development Method (DSDM), which focuses almost completely on processes—but they all have similarities.

A Little History

Although the term *agile*, as it relates to software development, dates from early 2001, agile methodologies have been in use for much longer. They used to be called iterative methodologies. Today's particular bunch were called lightweight development methodologies before the Manifesto for Agile Software Development was produced in February, 2001. (Seems someone didn't like being called a lightweight!)

> ***Individuals and interactions*** *over processes and tools*
> ***Working software*** *over comprehensive documentation*
> ***Customer collaboration*** *over contract negotiation*
> ***Responding to change*** *over following a plan*
> *That is, while there is value in the items on the right, we value the items on the left more.*[1]
>
> —Manifesto for Agile Software Development

A few years ago, people looked on agile development practices with great suspicion. The term was almost ridiculed in some circles. These days there is more respect paid, and these practices are making significant inroads. Most organizations I've worked with have flirted with agile methods. Developers are learning what works, either on their own projects or from experiences at other companies, and agile practices are spreading, often under the radar of the larger development organization.

Arguably, the wider adoption of agile methods reflects an underlying change in technology. This change began in the early '80s with the wide-scale introduction of personal computers. At that point, computing power was expensive and people's time was comparatively cheap. Experiments and prototyping were unknown. The ability to run hundreds or thousands of tests in a few seconds was fantasy. The idea of setting up and tearing down a SQL database in memory was absurd. A generation has passed, and that relationship has reversed. Development methods are finally catching up with the changes in technology, and the lessons learned from physical manufacturing in the '80s and '90s are also being felt.

While the various agile techniques are useful on their own, they have strong synergistic effects. One practice enables another to be used more effectively, so the payoff from using them in combination is often larger than for using them separately. I've tried to note such interactions in this chapter.

This chapter aims to show you what those methods are. I'll try to explain how they tie together. Some that relate to process won't be covered in this book, but those relating to tools will be. These are the same practices that are easiest to bring in the back door as a developer.

---

1. The Manifesto for Agile Software Development is available at `http://agilemanifesto.org/`. The authors are Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. It may be freely copied in any form, but only through to this notice.

Planning and Agile Development

Proponents of agile development methods often give short shrift to planning. I feel this is an overreaction to "big design up front" (BDUF), a practice often condemned by agile advocates. Planning is critical to any project. At the very least, the development team needs to know the broad scope and the intended form of the finished product; for example, a hosted solution is very different from shrink-wrapped software. It is important to defer coding until you have a basic grasp of what you are trying to build.

Agile methods aren't a license to go flying off in any direction. The admonition that an agile team should have expertise in the problem domain is often underplayed. This requirement for experience allows advocates to underplay the role of planning, because if you've built it once before, you've already invested the effort in planning once, and doing the same thing again is a slam dunk. In the more interesting and challenging cases, this is not true. At these times, it pays to sit down and think about the voyage you're embarking upon and the path that will take you to your destination. Failure to do this leads to failure.

I've been witness to an agile project that reached a dead end. The architecture the team had evolved couldn't cope with the new requirements. The team scrapped what they had done, and they launched into the process of rewriting the application. Rather than building in the desired architecture from the beginning, they dogmatically pursued the same evolutionary process that they had used the first time. Unsurprisingly, they ended up at the same dead end again. (To be fair, the outcome was foreseen by at least one Cassandra on the team, but she was ignored.) Eventually, they dug themselves out, but at the expense of quite a bit of developer time.

This leads to a conjecture that some recent work supports: agile development methods are excellent tools for producing locally optimal designs, but on their own they are insufficient to produce globally optimal designs. Development techniques are no substitute for a thorough understanding of the problem domain. You still need experts, and you still need to comprehend the big picture.

What Are Agile Methods?

Agile methods are a collection of different techniques that can be used in conjunction to achieve high software quality and accurate estimates of time and material with shorter development cycles. The laundry list includes pair programming, user stories, TDD, refactoring, simple design, rapid turnaround/short iterations, continuous integration, a consistent system metaphor, on-site customers, collective code ownership, continual readjustment of the development process, and believe it or not, documentation. The things relating to specific tools will be covered deeply in this book, but those relating to process will only be touched upon lightly.

The first insight into agile methods is that all software development is software maintenance. Feature development and feature maintenance are one and the same. Your software should always be functional. It may not have the full functionality of the finished application, but it should always be runnable and installable.

The second major insight is that source code is not a product. It is a blueprint for a product. The final product is packaged object code, or in some environments live code running on production hardware. What goes into the process is a design, and what comes out of the compiler or the interpreter is the product. A single project may in fact produce multiple programs or multiple packagings for different architectures.

This is a somewhat provocative statement, but there is a good deal of literature to back it up.

It seems less absurd when you examine how other manufacturing processes are becoming more like software. Once upon a time, design was only a small part of producing an ornate steel scrollwork grill. Production required days if not weeks of work. The pattern was drawn or scraped into the metal. The metal was heated, banged out, banged back into shape, and then reheated. This was done over and over, and the process proceeded inch by inch. When completed, each edge had to be filed down to smoothness.

The process has gotten faster over the last 200 years. Oxyacetylene torches easily make gross cuts, eliminating the need to heat and reheat. Angle grinders dramatically sped up the filing process. Plasma cutters made gross cuts even easier; cutting steel with a plasma cutter is like cutting warm butter with a steak knife, but it's still a manufacturing skill requiring hand-eye coordination.

Today there are computer-controlled cutting tables. You feed in a blueprint, load a sheet of metal, and press a button, and a few minutes later the grillwork is complete. Design has become the primary activity.

Writing software is not producing new features, but instead designing them. Similarly, rewriting existing software is really redesigning old features. Every software developer is also an architect. The two roles are one and the same. Producing software becomes an entirely automatic process that is often maintained by specialists (often referred to as *release engineers*).

So what are these methods about? Well, I'm going to start with one that I don't cover elsewhere in this book: pair programming.

Pair Programming

Pair programming is the most controversial of the bunch. Quite simply put, most programmers aren't that productive. Let's face it, programming is lonely, and we're social creatures. So programmers end up wasting half their day. They spend time reading e-mail, whether personal or company. They surf the Web. They fall into conversations with coworkers. To some extent, they are just trying to engage with other human beings.

Working alone with a computer has a strange effect on the human mind. The computer gives rewards and feedback, but it doesn't engage our limbic system—that layer of gray matter that distinguishes the mammalian brain from that of a reptile. It's what allows us to be caring parents and friends; it's what lets us feel another's pain or love. Frequently, programmers find themselves in strange state of mind; many programmers I know refer to it simply as *code space*. As a profession, we don't talk about it much. It's a place isolated from the rest of the human race. It takes time to come back from code space, often hours, and those are the hours that we have to spend with our families and friends.

Put two programmers together and their work becomes a social activity. They work together. They don't get stuck, they keep each other from being distracted, they don't go into code space, and they're happier at the end of the day. At worst, you haven't lost any productivity, and you've increased employee morale.

Pair programming arguably improves code quality. It works because it is a form of constant *code review*. Code reviews are a process in which programmers review and make suggestions about another developer's code. Code reviews have been found to consistently

decrease the number of bugs per thousand lines of code, and they have been found to be more effective at doing this than any other single measure.

Pair programming transfers knowledge between team members. In typical development environments, programmer-to-programmer learning is limited to brief exchanges. Those exchanges may be meetings in the break room, conversations in the hall, or formal meetings. In pair programming, the exchanges extend through the entire day. When people spend time together asking questions, they get to know each other. They lower their barriers, and they're willing to ask stupid questions that they'd otherwise spend all day researching.

A bullpen layout is often used with pair programming to facilitate exchanges between programmers. In a bullpen, there is no obstacle between you and the next person. There is nothing to stand between you and the person to your left when you need to ask a question. If you need help from someone who knows a given section of code, you can turn around and ask them.

A word often used in conjunction with pair programming is *collocation*. It refers to teams that are in the same location and can freely exchange ideas. It should be noted that a team that shares a single floor or building is not necessarily regarded as collocated. If there are physical barriers between programmers, then they are isolated. Walls impede the free exchange of ideas and information. The classic barrier is the cube wall.

This immediately brings to mind an office brimming with noise and distractions. Classically, these are death to programmer productivity, but in unpaired environments, programmers are trying to isolate themselves from other human beings. This isn't the case when pairing.

It's not that excessive noise and distractions aren't a problem with pair programming. It's that the programmers are engaged with their partners. As a species, we're very good at carrying on conversations with other people and ignoring our larger environment. When we're doing that, it takes much more to interrupt the flow of thoughts. Think of all the wonderful conversations that people have in restaurants and cafes. We can play immensely engaging games in such environments; chess and bridge come to mind. If the volume gets too high, then concentration will break down, but the environment has to get really raucous for that to happen. You want to work in a cafe rather than a night club, but that still leaves a wide range of environmental choices.

In any development organization, there is a huge amount of information stored in the heads of the developers. That knowledge will never be completely transferred to paper or bits. To do so would take more time and money than is available. Think of the difficulty of maintaining someone else's code when they are no longer around. What they could answer in seconds will take you minutes or hours to fathom out.

Code bases are full of questionable constructs. Pairing serves to spread the explanations from person to person. In order to understand what one person is doing, the other has to ask these questions.

Pairs are fluid. Programmers pair with different programmers every few days. This spreads the knowledge around. Knowledge spreads like a virus. One person knows something in the beginning. They pair with someone. Now two people know. They move on to different pairs, and now four people now know. The more pairings you have, the more it spreads. This protects the development group from the loss of any one programmer.

Caution Viruses spread like viruses too. Presenteeism and pair programming are a bad combination. If you're pairing and you get sick, then please go home and rest. The rest of us want to stay well.

All professions involve a large element of social learning. Lawyers and doctors have internships in which they engage with mentors and peers. In some medical schools, people work together in teams. Mathematicians, members of the classic loner profession, actually spend a huge amount of time in front of blackboards hashing out ideas together. Coffee fuels them, but it's usually flavored with chalk dust. Pair programming recognizes our natural strengths as social creatures and works with them.

User Stories

User stories are short descriptions of features to be written. They describe a small piece of functionality that can be broken down into tasks whose durations can be quickly estimated. They should fit on an index card (see Figure 1-1). They determine what we are going to produce. If it's not in a user story, we shouldn't be coding it. In a perfect world, user stories would determine every feature that goes into the software.

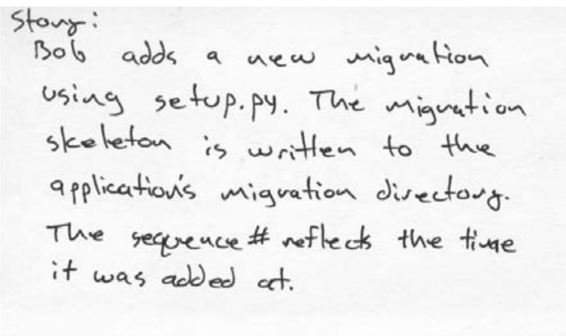


Figure 1-1. *Card with a user story*

User stories are produced in conjunction with the customer. They are a distillation of everything the customer knows. More importantly, they are the distillation of what the customer wants and what can be produced by the programmers. It is important for programmers and management to be involved in their creation, as they provide a technical check on customers' wild dreams. It is important that the dreams be those of the customer, though, as the programmers probably don't have as firm a grasp on the business problems as they'd like to believe.

Some user stories are produced just by the development organization. These relate to the internals of the software. They may describe a new data structure (say, a B-tree) or module (perhaps an object store) to be created.

User stories alone are often insufficient to specify the software's behavior fully. At those frequent points where clarification and elaboration are required, the on-site customer should be consulted. This results in a direct transfer of knowledge from the customer to the coder, bypassing the interpretation that would be imposed by more formal documentation.

The principal difference between a user story and a use case is that the effort required to complete a user story can be easily estimated.

The System Metaphor

The *system metaphor* allows you to talk about your design in a consistent and unambiguous way. There is just one way that the code, the developers, the managers, and the customers talk about the design. When everyone speaks the same vocabulary, meetings and discussions flow smoothly. (We've all been in those interminable meetings where everyone goes back and forth, confused over what a handful of words mean.)

The system metaphor should be used throughout the project. It should be used when discussing the project, in the user stories, and throughout the code base.

A variable or a function with a name that conflicts with the system metaphor should be treated as a bug. There is a children's game called telephone (also known as Chinese whispers). In it, a group of children sit in a line or a circle. The person at the beginning whispers a short phrase or sentence into the ear of the person next to them. That person repeats the phrase to the next person, and this continues until the phrase reaches the end, where the last person announces it. After the first person stops laughing at how far the phrase has been transmuted, they tell the group what the starting phrase was. Rarely if ever does a phrase make it through the line intact.

Naming things is a bit like the game of telephone. You look for a name that is close, distinctive, and appropriate. The further your base name is from the system metaphor, the further your new names are going to be from that. The meaning drifts, and eventually it becomes unintelligible to everyone except you. When someone returns to your code six months from now, they're going to be lost in the mess of unfamiliar terminology. That someone may be you. Therefore, names inconsistent with the system metaphor should be fixed immediately, and you should refer to the system metaphor when even slightly in doubt.

Automatic refactoring tools in a modern IDE (integrated development environment) help with changing names across the system. The process is almost magical. Chapter 7 will cover the use of refactoring tools in the Eclipse IDE.

On-Site Customers

On-site customers allow you to get feedback from someone who will actually use the product. Nothing tells you that a feature is off track as fast as a user saying, "What's that for?" When you have questions about what a user story means (of course, written in terms of the system metaphor), you can get an answer from the customer rather than guessing.

Few specifications are ever complete, particularly when the specification is written on 3×5 index cards. The job of the programmer is to turn rough specifications into precise and unambiguous instructions. Much of that translation is based on knowledge about how the computer works, but much of it is based on domain knowledge, and that knowledge belongs to the customer.

Having a customer available saves the programmer from having to make guesses about the domain. Making guesses is expensive. A wrong guess is a bug. It is a well-substantiated

observation that the further a bug makes it through the development process, the more expensive it is to fix.

The customer also serves as broad functional QA. She gets to see things along the way. She can catch situations where the developers and customer didn't communicate quite well enough. This prevents functional misapprehensions (bugs) from getting further into the development process.

The customer and the development team should communicate using the system metaphor. If they don't, then confusion can ensue, and the developers can produce the wrong thing.

Having the customer on site speeds up this process of interaction. The programmers don't have to wait for the customer to get back to them. If the programmers have to wait, then they lose their train of thought or have to switch tasks. This takes time. The easier it is for the programmer to get hold of the customer, the more likely he is to do it.

In the real world, we sometimes have disagreeable customers, or customers who feel that interacting with the development team is a distraction from their real work. Often they have been burned in the past. Short iterations of work help with this because the customer gets feedback. Good old-fashioned social engineering can help too. Never underestimate the power of a random thoughtful gift.

The more familiar the programmers are with the customer, the more likely they are to go to the customer for assistance. The more familiar the customer is with the programmers, the more likely she is to be happy to offer assistance. It is important that the customer and the programmers feel comfortable with each other. For this reason, among others, the customer should be included in the development group's work and social activities as much as possible.

Unit Tests

Unit tests are also called programmer tests. These are the tests that you write to verify the operation of your code. These tests aren't at the level of features. They are at the level of methods and functions. They allow you to check assertions during development, like, "If I pass in an empty list, does this function raise an exception?" Or, "If I pass in an unknown user, does this method return `False`?"

One of the joys of working with an interactive language is getting rapid feedback. You type a command, and you can see its output. You can verify that it's what you expected. You make a change to a program, you run the program, and you can immediately see the result. In noninteractive languages, you write a small program just to test the new functionality. You run the test programs, and in your mind you check that the result is what you expect. You conclude that the code worked, and you move on to the next chunk of code.

Unit testing formalizes that. You put all of those little test programs in a common place. Instead of checking the results manually, you write code to check the results. You package all of that up inside a harness to run the tests and report all their results. This way, you have a record of everything that you expect the code to do, and you can verify conformance at any point by rerunning the tests.

Unit tests should be pervasive. If your code isn't being tested, then you don't actually know that it is working. You trust that it is working, but you have no precise means of verifying this. Unit tests provide that means. They need to be pervasive because many bugs are nonlocal.

Code in one section of the code base can interact with code in another section of the code base. If you're not testing the entire code base, then you can miss these far-flung effects. These errors pile up, and soon you reach a situation that is referred to as "playing whack-a-mole."[2] The cause is often an underlying set of bugs elsewhere in the code. Pervasive unit tests let you see all of the problems at once.

Unit tests should be run after every change. Bugs interact, and the number of interactions doesn't increase linearly. One bug doesn't gives you one error. Two bugs don't necessarily give you two errors. They can give you four errors or more. Three bugs can give you eight or more. This is where the whack-a-mole situation comes from. Every time you make a change, you can introduce a bug. Running unit tests immediately after you make a change allows you to see a new bug as soon as it is created and before it has a chance to interact with other bugs. Fixing the bug is quick and cheap at that point. The more changes between runs of the unit test, the harder it will be to fix the resulting bugs.

Unit testing intersects with *regression testing*. Regression tests identify bugs that have been seen before and that have been fixed. Unit tests are a means of accomplishing regression testing, but they serve a larger purpose. When writing new features, the primary goals of unit testing are confirming expected behavior and identifying bugs resulting from the new functionality. Unit test runs should be fast. If test runs take too long, then programmers won't run the tests as often. As noted previously, the less often the unit tests are run, the more errors will accumulate. The more errors that accumulate, the more expensive they are to fix and the more time is spent fixing them.

Unit tests shouldn't be too pervasive. Each unit test has a maintenance cost associated with it. If you change the code it tests, then the unit test will need to be changed. If more than one unit test examines a section of code, then all of the tests will need to be updated. This can quickly become onerous, and once it become onerous, developers will tend to stop maintaining the tests.

Unit tests shouldn't traverse code outside of the unit being tested. When this happens, a change in one package can cause a cascade of unit test failures. All of these tests need to be fixed before the test suite runs correctly again. A simple trip next door becomes an expedition. Finally, limiting the number of tests and depth of their reach promotes fast unit test runs. There are fewer of them, and they are doing less work.

In Python, we have many tools to assist with writing unit tests. Two of the most common are unittest and Nose. I'll show these to you in Chapter 6. unittest, which is older, is based on a classic design called xUnit. Knowledge of unittest will carry over to many other languages. Nose subsumes unittest, and provides a mechanism for running many tests. I'll show you how these test frameworks can be automatically run both in the local development environment and by automated build systems.

Mock object frameworks allow you to separate packages and classes from resources that they depend on. Limiting the scope of your tests is much harder without them.

Test-Driven Development

Test-driven development (TDD) turns unit testing on its head. All programmers know what they expect code to do before they write it; TDD makes these expectations concrete. It has a

2. Named after the game where moles pop out of holes, and you whack them with a hammer as fast as you can, but as soon as you do, they pop up somewhere else.

unique yet satisfying rhythm. You write the test. You run the test, and it should fail. You write the fragment of code being tested. You run the test, and now it should succeed. At every step, you're performing experiments to verify that your code operates as you expect.

TDD is done at a very fine granularity. You don't write all of the tests for the program. You just write the test for the next few lines. Many of these tests will be thrown away in the end because they test intermediate states in the program's evolution. Writing the tests at a fine granularity results in methods that are very small and possess very limited functionality. The same happens with classes.

At every step of the program's development, you have to think about how you are going to test the results. The data on which each method depends must be passed to that method as part of the test. Each data transfer tends to take effort, so you tend to create fewer of them. You have to test all the side effects, so you produce fewer of them. You have to test the classes in isolation, so you produce lightly coupled classes.

In general, big nasty methods and classes produce big nasty tests. Writing nasty tests is painful, so developers don't do that. Writing cleaner code becomes the path of least resistance. Simply put, designing for testing forces us to produce better code. And because we write the test before writing the code, we end up with close to 100 percent testing coverage.

I took the opportunity to do an experiment at work once. I was working on two sets of scripts—one was a new script, and one was maintenance on one of our nastiest, longest-running, and most problematic pieces of code. I wrote the virgin scripts without TDD. I made the overhaul of the nasty scripts using TDD. The nasty scripts ended up working fairly painlessly on the first try, and subsequent changes were utterly painless. The new code wasn't bad, but it wasn't nearly as robust. I would hate to see what it's like today.

Refactoring

Refactoring is the practice of simplifying and clarifying code at every opportunity. It focuses on changes that alter the structure of the code without altering the meaning. I find refactoring quite fun, but it's dangerous if you don't have unit tests in place. Without unit tests, you don't know if you've unintentionally altered the meaning.

In some sense, refactoring is one of the most well-understood areas of software development. It is described extensively in the literature. Refactorings have names and precise definitions. Some refactorings are done with an eye to improving readability, some are done with an eye toward removing redundancy and duplication (known to some as "the death of code"), and some are done to improve modularity. Most are limited in scope.

Refactorings should be limited to small localized changes. Large refactorings tend to affect many files, classes, and methods, and entail changes to many unrelated pieces of code. They can break everything they touch, and they require rewriting many unit tests. I have personally seen a large one-shot reorganization go quite awry. It consumed over a week of a medium-sized development organization's time. Large reorganizations across the entire code base are risky.

Refactoring is the daily bread of agile programming. It's the primary tool in producing simple designs. Whenever we encounter a chunk of code that doesn't smell right, we refactor until the smell goes away. I'll cover refactoring in more detail in Chapters 6 and 7.

Refactorings are so well understood that many are downright mechanical in nature. As such, they can be done with the assistance of tools. Most IDEs these days include tools to help with refactoring. The tools are much better established in statically typed languages, since

more of the structure and semantics of the code can be inferred, but tools for Python are getting better. Eclipse running Pydev, which I'll show you in Chapter 2, has some of the best refactoring tools available for Python.

The presence of automatic refactorings is one of the primary reasons for switching to an IDE; this alone nearly drove me to abandon Emacs for Eclipse. At first, you're likely to be a little put off by using them. They seem a bit clunky, but they're worth persisting in and learning how to use correctly.

Simple Design

Simple design is about fulfilling the requirements of the user stories and no more. Your code is the design, and the design should be as simple as possible, but no simpler. Simple design is not an admonition that design is bad, or that some up-front design isn't necessary. It is an admonition that *too much* up-front design is bad. How much is a matter of judgment and experience. Simple design encompasses a number of principles, many with cute acronyms.

The first principle is *don't repeat yourself (DRY)*. There should be one and only one source of truth for any fact or assertion in your program. If there is more than one, then someone will eventually end up changing one and not the other. And if they don't catch that, then it can lead to mysterious bugs, and in the worst cases it leads to architectural mistakes. Replicated functionality is the death of good code. When you find duplicated code, refactor!

The second principle is *you're not going to need it (YAGNI)*. Useless code is expensive. Every little feature and whiz-bang gizmo has a cost associated with it, and that cost carries into the future forever. It makes the code base complicated. It derails your delivery schedule. It keeps you from going running at lunch, makes you late picking up the kids, and delays you from cooking dinner for your wife. Is that cool generalization of the argument-processing code you might use the next time you reuse this code really worth it? Probably not. You can write it when you actually need it three months from now. This afternoon, you've got better things to do, like completing that user story and then playing in the snow with your sweetie.

The third principle is *better raw than wrong*. Simple, direct, blunt communication is better than anything else. You should make your intent clear. Don't mince words. Don't look for the most elegant way of saying things. Just say things simply and directly.

The fourth principle is *do the simplest thing that could possibly work*. You can make it more complicated and robust later. That clever code is generally harder to maintain. It takes more work to write, and it takes more work to test. It's a waste of time. Make the unit tests pass, and get on to the next feature. Be Hemingway, don't be Melville (Melville was paid by the word).

Collective Code Ownership

Collective code ownership is about who fixes what. It is based on the idea that individuals should not maintain ownership of sections of code. It is one giant code base. When there are problems, everyone should feel empowered to go right in and fix the code.

Having one or two people lording control over a code fiefdom transforms it into a bottleneck. We've all experienced this: it may be good code, and it may be bad code, but it's not our code. At one time or another, we've found a bug in someone else's fiefdom. Or we've needed a new feature. Or we've needed to interface to their fiefdom in a way that just wasn't intended. We need them to make a change, and they're not available. They may be busy on a critical project. They may be on vacation. They may not like us, and they may be exercising control.

The only choices are to wait for them to become available or to fix the problem from the outside. I've seen far too many warts accumulate because people didn't want to dip into a chunk of code. Many lines of logic could be created to handle erroneous results from a buggy package when a one-line fix would be sufficient within the package itself. Sometimes arguments have to be converted to strange formats and then converted back when it would be easier to add a new input type to a case statement inside the module. Or even worse, an entirely new framework has to be adopted because nobody wants to fix the performance problems inside somebody else's package. (Yes, I actually did see this happen. And it was a two-line fix that needed to be made.) In every case, fixing the problem from the outside is the wrong solution.

If everyone is making improvements to the code base as a whole, then the code base shouldn't go rotten. When a developer finds something that smells, they are authorized to find that smell and fix it. They are not just authorized to do this; they are expected to. At every opportunity, they are expected to refactor stale code that affects their current task.

Other agile practices facilitate collective code ownership. Pair programming breeds familiarity with the code base as a whole. It forces programmers to give up sole control of their fiefdoms. Unit tests allow you to make changes with confidence. Frequent refactorings of the code base keep it from going smelly, and this encourages developers to continue to maintain it.

Short Iterations

Short iterations serve multiple purposes. They allow you to deliver a working product to your customer at regular intervals. They allow you to see how accurate your estimates are on a regular basis. Finally, they give you an opportunity to regularly reexamine your development processes. You get to plan for the future and look back at the past while everything is still fresh in your mind. Note that shorter iterations are not necessarily better. Time spans in the range of two weeks to one month seem to be values that people work with successfully.

Producing a functional product on a regular basis allows your customer to judge the outcome. Since you've produced a small number of features, the customer can examine all of them quickly. The review's coverage is complete. The review is kept short, so the customer is fresh and observant all the way through. His feedback is likely to be detailed and thorough. Miscommunications about his intent will be caught. He won't get to the point where he'll say, "*O$#&! it, it's good enough." The customer will be more satisfied with the final product as a result.

With a large number of features, some will invariably be skipped, and if they are not, the review becomes a painful slog. People are likely to get tired and bored, and tired and bored people get sloppy. Sloppy behavior results in inadequately reviewed features. Design bugs will be missed, or features won't be quite what was intended. These inadequacies will progress further into the development cycles, and may eventually be released in production software. The further they get into the development cycle, the more expensive fixing them will be.

If the iterations are short, there will never be a grand moment of shock for the customer when she asks, "What on earth did you build? That's not what I've asked for." (It's even less likely if your customer was on-site and interacting with you.) There may be times when the customer says, "That's not quite what I asked for. How about doing it like this?" and she continues to describe what she needs.

The review should be a pleasant experience for everyone. Keeping the number of new features to a minimum helps with this. The less pleasant a review, the more likely it is be put off, and the less effort is likely to be put into it. Keeping down the number of reviewed features keeps the review short and pleasant, and that keeps the customer happy.

In a classic waterfall process, estimates are made months out. On a ten-month project, a 20 percent underestimate is a two-month delay. This is the difference between finishing at Halloween and being stuck at work for Thanksgiving, Hanukkah, Christmas, and New Year.

Agile projects make short-term estimates. On a two-week iteration, a 20 percent underestimate is only two work days. After the iteration is done, you get to produce another set of estimates. The causes of your delays will have happened days ago. The events that are going to happen in the next few weeks are likely to be known. Your estimates should be more accurate because of this, and you'll have many opportunities to learn from your misjudgments. Fewer inaccuracies should creep in, and you'll have many chances to tune the estimates before they damage the project.

Compare this with long-term estimates. The causes of inaccuracies are likely to have happened months ago. It will be hard to remember them when trying to learn how to estimate for the next project. You're not going to get many opportunities to learn how to estimate either. Many unforeseen events are likely to happen over the course of the new estimates, too. On a one-year project, it's likely that someone on your team may meet the love of their life, have a midlife crisis, or have a close relative die. These will all impact the project estimates.

Having accurate short-term estimates allows development leads to combat scope creep. When new features are requested, their effects on the schedule can be quickly and accurately determined. Management can be presented with this information, and they can be given options as to which features will have to be dropped in order to accomplish the new tasks. If estimates have been fed back to management throughout the development process, then they will have trust in the truth of these judgments. With long time spans between estimates, it is too easy for management to lose touch with the realities of software development and the direct effect that their actions can have on the process.

Iterations should produce a full product. All aspects of the production process should be exercised. There should be a working build, the build should be packaged, and that package should be deployed. It should go through all release tests, including load testing. If this has not happened, then there are surprises waiting for you. In places where I've worked, these have included

- A product that can't be deployed to production.
- An online product that only supports a few users on massive production hardware.
- A massive online system in which nobody asked the question, "How do we bill customers?"

All of these resulted in massive employee overtime and schedule slips. They were huge emergencies, and they reflected very poorly on the development organization. Had any of these projects used short iteration processes that moved from development through to production deployment, then these issues would have been caught.

Producing short iterations depends on automation. This automation chain runs from the developer's desk to the production facility. Human interaction should only be required at points where human judgment is required.

Manual processes should be avoided, because they are error prone. We're not built for doing the same thing over and over again. We get bored, and when we get bored we make mistakes. Mistakes take time and effort to find and fix. This causes unpredictability in estimates and scheduling.

Manual processes result in nonconformity. Each person invariably interprets instructions in a slightly different way, and every person makes mistakes. The result is that manual processes always introduce varying results. A netmask may be recorded incorrectly, the software may be copied to the wrong directory, or files might be named in a way that subtly invalidates chosen conventions. Automation must be able to cope with this variation, and this is a daunting task. Some might say that it is an impossible task. The amount of work involved in making the software flexible enough to handle these variations is often far larger than simply automating the manual processes.

People are slow. People are unpredictable. People are limited in the amount of work they can perform. People don't scale. Automation is how we get around these issues. Automation itself has the potential to go horribly wrong, though, and this is why it has to be exercised as part of the build process. The same automation should be used in development as is used in production. That way, the build process itself verifies the integrity of the automation.

Much of this book is about building that automation chain. It shows how your development environment can automatically run your tests. It shows how to build your changes automatically, how to package your application, and how to upgrade your database schemas repeatedly and reliably. All of these facilitate short iterations.

Continuous Reflection

Continuous reflection is the ongoing analysis of the development process. The development process is not something that is set in stone. At every opportunity it should be subjected to scrutiny and refined. There should be just enough process and no more. This reflection is often done at the end or beginning of an iteration.

Process exists to coordinate activities between people. It has benefits, and it has costs. Within an organization, it protects some parts from abuse by others, setting boundaries and responsibilities. It provides a framework for communication, and it's also the communication itself, allowing large numbers of people to work together in ways that they might otherwise not. At the grandest scale, there are international treaties and processes for dispute resolution that coordinate certain activities for billions of people. At the other end are simple rules for individuals greeting one another on the street.

Within a development organization, processes often set expectations between management and development. They define who is doing work, what sort of work they are doing, how that work will be performed, and when the work will be done. This gives a degree of predictability and transparency.

Process often exists to coordinate activities between groups. It lets QA know what development intends to deliver. It lets development know when QA needs the first release. It lets release engineering know when it should schedule a deployment test, and it lets system operations know when they should schedule the deployment. In these cases, it replaces clear communication and personal relationships between groups. With large groups this is necessary. With smaller groups it may not be.

It also gives each party a means of lowering their risks if something goes wrong. Even though a project may fail, the documents mandated by the process can be used to show that

expectations were met. When this happens, something is wrong with the organization. The process has become more important than actual accomplishments.

Too much process is maddening. People can see when formal processes impede work rather than facilitating it. They begin to resent the time they spend on the process, and it becomes a point of frustration. If the process takes away responsibilities, then they often feel powerless. They will feel even more powerless if they can't alter or bypass the process. There is little in the world that is as devastating to mental health as a high-stress environment in which people feel they have little control.

At best, the onerous process will be circumvented and become nothing more than a small waste of people's time. At worst, it will become a point of contention that will breed employee dissatisfaction and lead to turnover.

The only thing worse than too much process is no process at all. When no expectations are set, when no procedures are defined, or when anything goes if you ask nicely enough, the development process can run off the rails. People head in different directions. Individuals become points of control, and losing them can then be devastating to the organization.

Agile development processes try to find the sweet spot between these two extremes, and continuous reflection is the means. At the end of every iteration, the development teams look at their processes. If something is giving benefit, then it can be kept. If it has more cost than benefit, then it can be abandoned by the group. If there is not enough process, then the minimal amount of process necessary can be self-imposed. There is clear discussion so that everyone knows why the process exists and why it is retained. The team members are left with a feeling of control.

Agile teams feel they can get away with less process. The focus on automation reduces the number of people needed. That reduces the need for coordination. The frequent feedback from the short iterations increases visibility into the development process. It also produces more accurate estimates, increasing predictability. Pairing and collocation reduce the need for formal exchanges and meetings. Thus the various agile processes compensate for the reduced process load and, in doing so, actually accomplish some of the very goals that more formal processes strive to achieve.

Continuous Integration

Continuous integration is one of the most general practices. Code lives in the source repository. The longer your code is away from this repository, the further it will diverge from the repository version. When the code is merged back in, you will find bugs. The odds of any two changes conflicting go up nonlinearly, so the longer you wait, the more conflicts you will find. Resolving conflicts will become more painful, and more unit tests will have to be rewritten.

The solution is checking your code into the repository as frequently as possible. Every hour your code is out of the repository is another hour in which it can diverge from other developers' work. Every day your code is out of the repository means another day's salary that has been sunk into untracked changes. Should your machine crash, those changes will be lost, and that money and (more importantly) development time will also be lost.

These submissions should not break the build, nor should they break the application. Developers need to have a way to verify that they haven't broken the build. They need to do this in their development environment. The obvious choice is running the build and unit tests locally. Optimally, this should be the same build that is run for production, as every difference is a possible source of failure. The code should always compile, and all of the unit tests should succeed before code is checked into the source code repository.

There should be one source code repository for a project. All of the project's code should be checked in here. All artifacts necessary to produce the build should be included, too. This includes build scripts, tool scripts, properties files, installation scripts, third-party libraries, tests, and tool configurations (e.g., IDE configuration files).

The repository itself can enforce certain policy decisions. Submission triggers that validate code can be placed in the repository. If a file is not successfully validated, then the code submission will fail. Frequent validations include style analysis and syntactic analysis. With Python programs, style analysis checks frequently verify correct whitespacing. Syntactic analysis can be as simple as verifying that the file parses successfully. While the checks are being performed, submissions to the repository are blocked, so submission validation does not extend to significant functional checks.

Builds should happen automatically and as frequently as resources will allow. This is done to discover bugs as soon as possible. At the very least, builds should be run nightly, but with today's computing resources, there is very little reason not to run a build whenever new source code is added to the repository.

No human intervention should be necessary to go from source to finished and tested product. The build system should check out a clean copy from revision control, and then build from it. This ensures that the build does not depend on previously generated artifacts, and it tests that the build can be done on a machine other than the developer's desktop. This also goes a long way to ensuring that any developer can sync the code tree down to a new machine, issue a single build command, and have the build succeed. This allows desktops to be replaced quickly in case of failure, and it helps new developers on a project to come up to speed quickly.

The build should test all components, construction of the database, and initialization of any external services. The build must run all of the unit tests. Any unit test failures should cause the build to fail. I'll argue that a minimal set of functional tests should be run, but these may be restricted to certain kinds of builds. Often these are referred to as official builds.

Build failures should be quickly communicated to the team so that they can be quickly fixed. Typically, this entire process will be done on dedicated build machines, so a remote notification system should be used. This might be mail, chat, or some external means of notification (such as a lava lamp or a siren).

Much of this book focuses on continuous integration in Python. The package Setuptools forms the core of a replicable build system in Python. It provides dependency management, building, packaging, and a test harness. The testing package Nose will actually run the unit tests. The unit tests will run both from the build and from within the Eclipse IDE. A set of custom scripts will handle database management. The repository we'll be using is called Subversion. We'll validate Python source with Subversion triggers. Finally, we'll set up a build server using Buildbot.

Documentation

Documentation should be minimal. It should be limited to that which is necessary to ensure that participants in the development process can communicate. You should probably keep your system metaphor available somewhere. You should have the documentation necessary so that your system can be used and maintained.

You do not need extensive design documentation. The code is the design. You don't need extensive commenting. The code and unit tests should be clear and readable. Extensive design

documentation is the pinnacle violation of DRY. The impression of the design invariably falls out of sync with the reality of the design.

These practices have a common focus. They are about predictability. They are about minimizing waste, both in process and in design effort, and continuous feedback to identify the sources of waste.

While it's important to understand how these techniques fit together, we're not going to be looking at all of them. Our focus is going to be those agile techniques that are abetted by supporting software. These areas are continuous integration, TDD, unit tests, refactoring, and simple design. None of the products we're looking at are commercial. Like Python, they're open source of one sort or another. These products (and the corresponding chapters they're covered in) are as follows:

- Eclipse (Chapter 2), an IDE
- Pydev (Chapter 2), a Python Eclipse component
- Subversion (Chapter 3), a revision control system
- Setuptools (Chapter 4), a build harness
- Buildbot (Chapter 5), a continuous build system
- Nose (Chapter 6), a test runner
- pMock (Chapter 7), a mock object framework
- PyMock (Chapter 7), another mock object framework
- SQLObject (Chapter 9), an object-relational mapper
- SQLAlchemy (Chapter 9), another object-relational mapper
- JSUnit (Chapter 10), a JavaScript unit testing tool
- PyFit (Chapter 11), a functional testing tool

Summary

In this chapter, I have introduced the methods that constitute much of agile development. You've seen how they tie together and how they assist each other. Some of these methods relate to process, but many have more to do with concrete programming practices. Some are strictly developer tasks, and others are often seen as part of release engineering. By now, you probably realize that many agile methods aren't so alien. If you're a professional developer, you've probably used several of them, quite possibly automated builds and unit tests.

Much of what you will learn here can be taken back to work. If you decide to do that, then be careful. The word *agile* scares some people. Even though the techniques are hardly earth shattering in their novelty, the term has become a four-letter word in some places; it turns people off immediately.

If you're a developer, and you decide to bring an agile method into work, then treat it like a strange and somewhat skittish pet. Bring it into work, show it around your desk, let it become comfortable with its immediate surroundings, and let it interact with the local

environment. If it seems to be happy with you and your code base, then introduce it to some of your more inquisitive coworkers. Let them play with your friend. When it becomes at home and comfortable, see if it expands its home range on its own. Once it's comfortably ensconced in the new home, then bring in one of its friends in a similar way.

If you're in management, it is often best to lead your team gently into the new practices. While many find the prospect exciting, some will be skeptical. Saying, "We're going agile now" is a good way to lose face. Find a developer or team that is interested in trying out the techniques. Choose a practice with a low barrier to entry and a high payback for the effort, and get the developer or group to try it out as a pilot. There will be teething problems, so don't talk it up until the practice has proven itself. Changes are always most successful if the developers believe they instituted the changes themselves. If the practice succeeds, make sure that the developers involved get the credit. If it fails, take the responsibility yourself.

I use the command line for much of the material in this book but, whenever possible, I try to present the same material within an IDE. too. You can certainly do agile development without an IDE, but there are some tasks that are far more difficult. One that springs immediately to mind is refactoring.

In the next chapter, I'll show you the Eclipse IDE and how to set it up with the tools that we'll be using throughout the rest of this book.

CHAPTER 2

The IDE: Eclipsing the Command Line

There are two main ways you can work with Python: through the command line or through an IDE. Both have their distinct advantages and disadvantages. I'm not going to give short shrift to either, but this chapter is mostly about using the Eclipse IDE to work with Python.

To truly take advantage of agile development methodologies, chances are you're going to want to use an IDE. IDEs offer a range of features that are at best poorly implemented in command-line tools. A short list includes a wealth of code navigation features, intuitive autocompletion, refactoring support, revision control integration, automatic builds, polished debugging, integration with unit testing tools, language-aware editors, jobs contexts, and ticket system integration. All of these features are useful, but when integrated they deliver more than the sum of their parts.

I've chosen the Eclipse IDE for the purposes of this book. Eclipse has a number of things going for it. It's free, it's widely used (I've seen it in most companies I've visited), and it's available on just about any platform you could desire (I'm almost certain that it will run on HP calculators these days). It's extensible, and it has Python support. It also has a few features that others lack, notably a job system called Mylyn (formerly known as Mylar—the name has changed, but the functionality remains the same). It also has a plug-in architecture that allows users to write custom extensions.

WHAT IS A JOB MANAGEMENT SYSTEM, AND WHY DO I NEED ONE?

A large project will have hundreds or thousands of files. Typically, each task that you work on will have only a small subset of these open at a time. You'll be working on a few tasks on and off, and you may have to go back and forth between them. These tasks will have few if any files in common, so you'll either end up with many tens of editors open or spend a great deal of time opening and closing them to keep your workspace clean.

Even though you'll have many files open, you'll typically only reference a few classes or methods. Eclipse offers navigation panes to help locate specific program elements such as files, classes, methods, and functions; but when working with a large program, finding a specific element is still troublesome. It's a bit like finding one particular piece in a big box of Lego parts.

Mylyn addresses both of these issues. It defines *tasks*, which represent units of work. These tasks may be defined locally, or they may reference external tickets in a defect-tracking system such as Bugzilla or Jira. One task is active at a time. Mylyn tracks program elements as you work on them, and it associates these with the active task. This collection of elements is referred to as the *context*.

When you activate a new task, the current task is deactivated. All of the editors associated with it are closed, and all of the editors associated with the newly activated task are opened. It also restricts navigation panes to show only those packages, directories, files, classes, methods, and functions that are associated with the active task. This filtering is turned on and off on a pane-by-pane basis by toggling an icon in the pane's menu bar.

In addition to context, tasks also have associated planning information. This information includes priority, severity, and scheduling data such as expected delivery date. Although not perfect, Mylyn is a major innovation in IDE interfaces.

A large ecosystem has grown up around Eclipse and its plug-in architecture. There are plug-ins for just about every imaginable task. Examples include style checking, C and C++ compilers, database tools, revision control, web server integration, spelling, and hundreds of others.

The plug-ins I'm interested in showing are Mylyn, Pydev, Pydev Extensions, Subversive, and SQLExplorer. Eclipse is natively a Java development environment. Pydev is a free plug-in that teaches Eclipse to work with Python. Pydev Extensions is a commercial addition to Pydev that does even more, and I find it well worth the small price. Subversive allows Eclipse to work with the Subversion source code repository, making simple revision control tasks almost transparent. Finally, SQLExplorer lets you browse and query databases and their schemas. I'll come back to Subversive in Chapter 3 and SQLExplorer in Chapter 9, but for now we're going to look at Pydev.

From all the glowing statements I've made, you might get the impression that Eclipse is the only game in town. It's not. There are a number of other good choices out there. Most are not free, but they are comparatively low cost. The ones worth noting are Wingware's Wing IDE (`www.wingware.com/`), ActiveState's Komodo (`www.activestate.com/`), and in the Microsoft Windows world, the .NET development environment. I wish I could include JetBrains IntelliJ IDEA in here, but until someone produces a mature Python plug-in for it, we're out of luck.

Eclipse feels a little clunky compared to these others. It has all the features of its competitors, but the interface is a little less polished. Having made my disparaging comments, I'm still going to use Eclipse. There are mind-boggling numbers of plug-ins available, and with the right ones, it does what you'll need it to do.

Installing Eclipse

Having decided on Eclipse, the first step is to get it onto your system. Eclipse lives at `www.eclipse.org/`. The download URL (as of this writing) is `www.eclipse.org/downloads/`. You'll want to get the package called Eclipse Classic. This gives you the kitchen sink.

■ **Warning** The Pydev plug-in can't cope with spaces in the workspace path, so you should ensure that it does not contain any.

Start Eclipse once you've downloaded and installed it. When Eclipse starts up, it will ask you for a workspace location (as shown in Figure 2-1). This is the directory tree in which Eclipse will store all of its data. The workspace isn't a shared resource, so it should be within your home directory. It should be easily backed up, easily remembered, and quickly accessible from the command line. I personally choose the default. (On my Mac, that's `/Users/jeff/Documents/workspace`.)

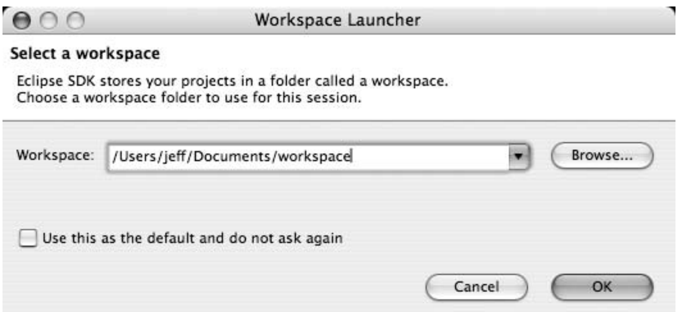


Figure 2-1. *Selecting the workspace root*

Eclipse will grind for a while as it starts up and creates your workspace. It will bring you to the initial landing page, shown in Figure 2-2. This page only shows up until you've created a project, so you won't see it very often.

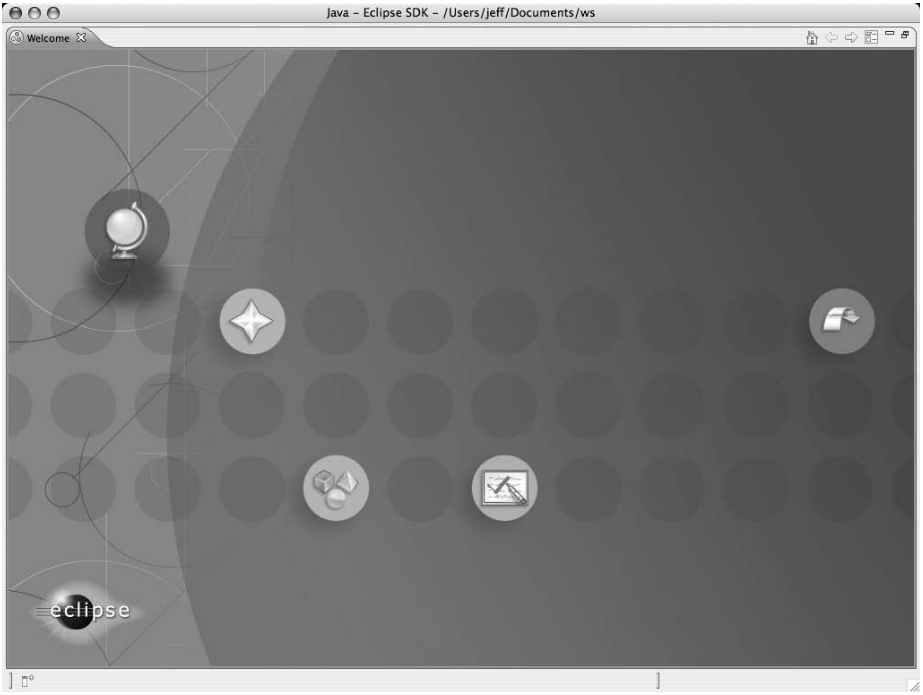


Figure 2-2. *The startup screen, which you probably won't see again*

On the right side is an arrow heading into the distance. This takes you to the workbench. The workbench is where everything happens in Eclipse. It is shown in Figure 2-3.

Now that you can see what Eclipse looks like, it's time for some explanation of the major pieces. Within the workbench are *perspectives*. Perspectives are dedicated to some particular kind of task. Examples include Java development, debugging, source code repository, and plug-in development. The perspective is indicated in the tab at the top right of the window. By default, Eclipse opens into the Java perspective.

The little glyph to the right of the Java indicator allows you to select a different perspective. Once you open a perspective, it stays active, and its icon stays in the tab. By default, the tab is a little small. As you switch between perspectives to perform different tasks, it will quickly fill up, making it harder to switch back to previously used perspectives. You can remedy this by grabbing the tab's left edge and dragging it further to the left, making room for additional icons. This demonstrates a general principle of Eclipse's interface: pretty much everything can be moved or rearranged.

The smaller panes are called *views*. Views do specific functions within a given perspective. Examples are showing console output, viewing outlines, and editing files. You can rearrange the views to your heart's content. You can do this by grabbing the tab and dragging it to another spot within the perspective, or you can drag it onto the desktop, and it will become a free-floating window.

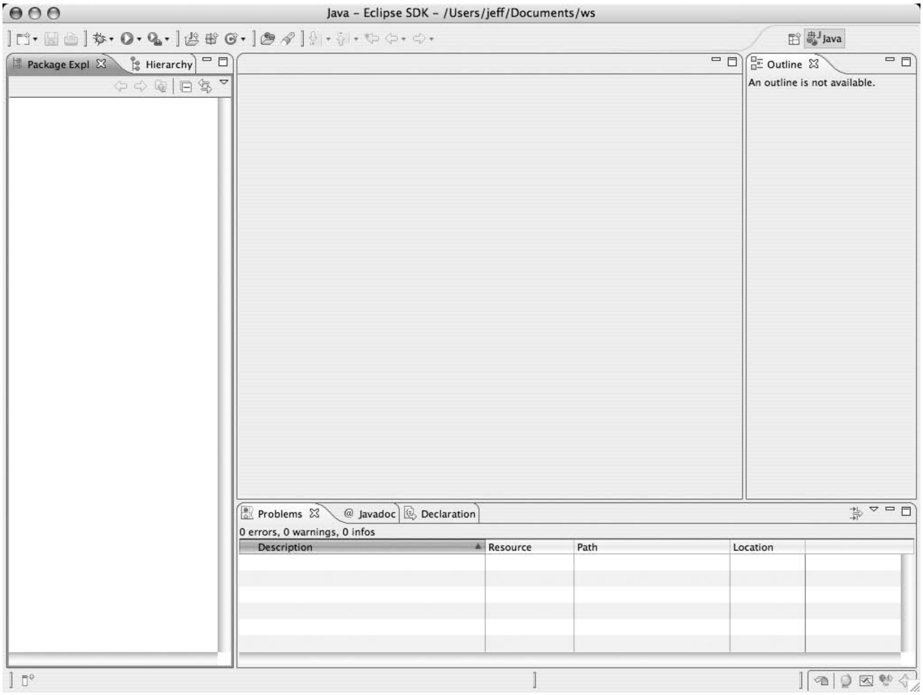


Figure 2-3. *The empty workbench*

We're working in Python, so the Java perspective isn't going to do us much good. We need a plug-in that teaches Eclipse to work with our language. This plug-in is called Pydev. We could hop into installing Pydev right away, but it has some features that depend on another plug-in you're eventually going to want to use. That plug-in is called Mylyn. It is absolutely generic in its installation, so it makes a good example. The process for installing Mylyn and Pydev is much the same.

Installing Plug-Ins

Eclipse plug-ins are published on little web sites known as update sites, and are very easy to install. You give Eclipse the URL for an update site, and it sucks down the plug-in and installs it. Be careful though—don't confuse the web site for a plug-in with its update site. The web site for Mylyn is `www.eclipse.org/mylyn`, while the update site is located at `http://download.eclipse.org/tools/mylyn/update/e3.3`.

Start the installation process by selecting Help ➤ Software Updates ➤ Find and Install. This brings up the Install/Update window, shown in Figure 2-4.



Figure 2-4. *The first screen of the Install/Update wizard*

Select "Search for new features to install," and then click Next. The resulting dialog, shown in Figure 2-5, lets you add a new update site.

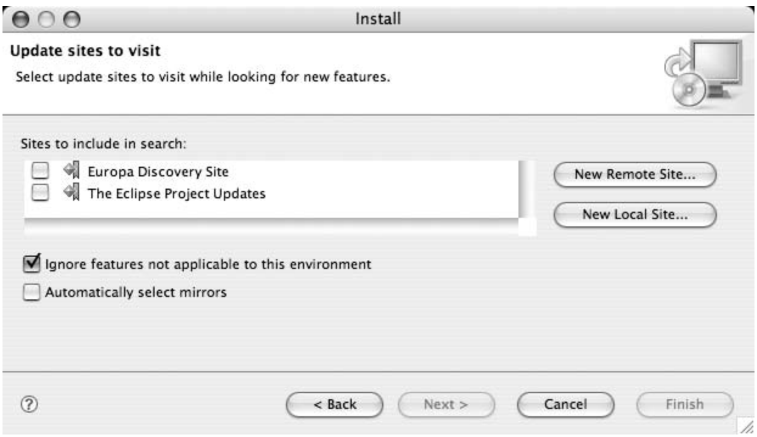


Figure 2-5. *Choosing features to install*

Click the New Remote Site button on the right. This takes you to a dialog with two fields (shown in Figure 2-6).

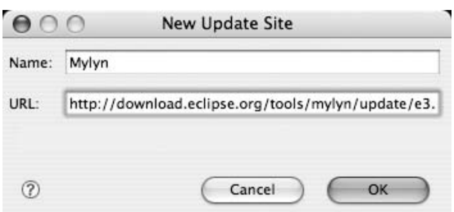


Figure 2-6. *Creating a new feature*

Fill them in as shown in the figure, and click OK. (Note that the URL is bigger than the window and has been cropped a bit.) You'll be taken back to the Install window, as shown in Figure 2-7.



Figure 2-7. *Choosing the newly created Mylyn update site*

The Mylyn site should be highlighted. When you click Finish, Eclipse will download all of the selected updates.

Eclipse will grind for a moment or two while it queries the update site. Assuming that it is successful, it will bring you to the Search Results screen, shown in Figure 2-8.

It's possible to have more than one result, so you have an opportunity to select which plug-ins you want to install (and which parts of which plug-ins if you so desire). Select Mylyn, as shown in the Figure 2-8, and click Next. This takes you to the Feature License screen, shown in Figure 2-9.

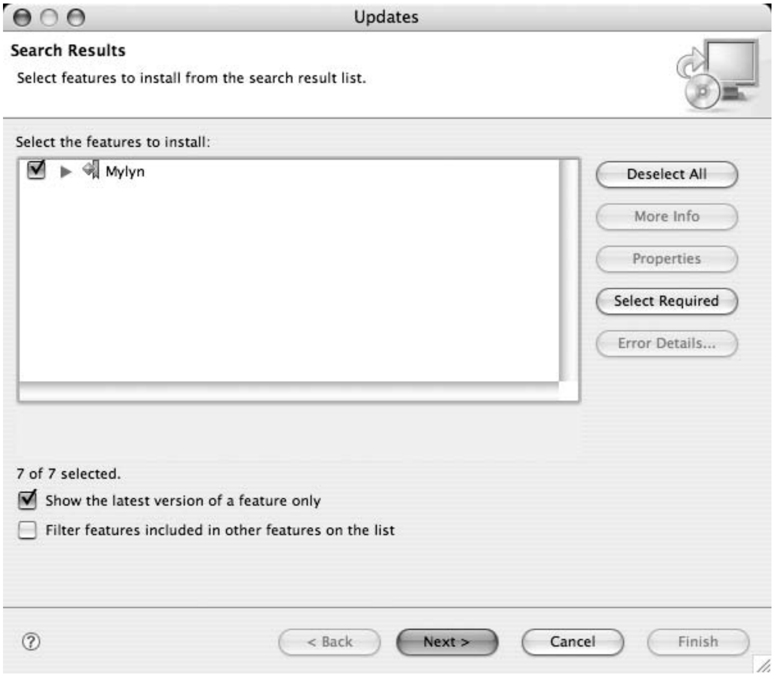


Figure 2-8. *Choosing from the (only) returned update site*

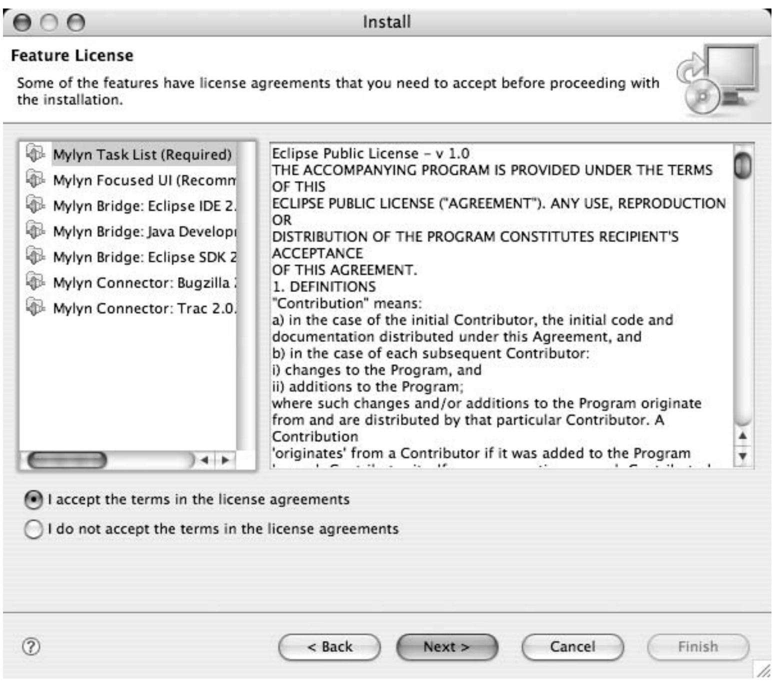


Figure 2-9. *Accepting the license agreements*

Read the license agreements, and if you agree, click "I accept the terms in the license agreements." Then click Next. That will take you (finally) to the Installation screen, shown in Figure 2-10.

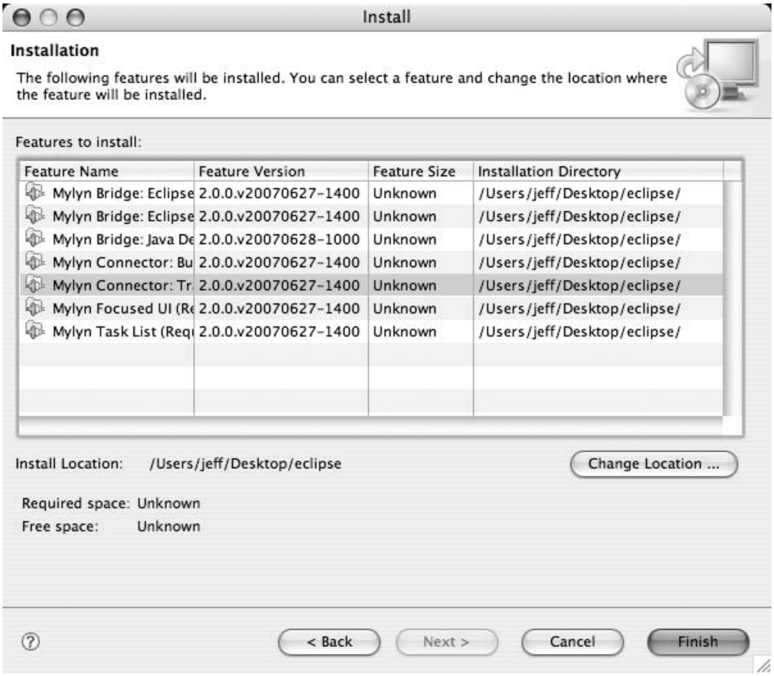


Figure 2-10. *The components will be downloaded.*

The Installation screen gives you a chance to change where the components are located. Don't be tempted. Click Finish, and Mylyn will start downloading and installing. You'll see the dialog shown in Figure 2-11.

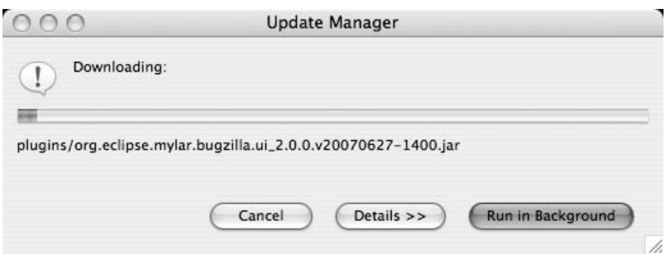


Figure 2-11. *The update downloading*

The installation will progress for a while. At some point, the process will halt, and you will see the window shown in Figure 2-12.

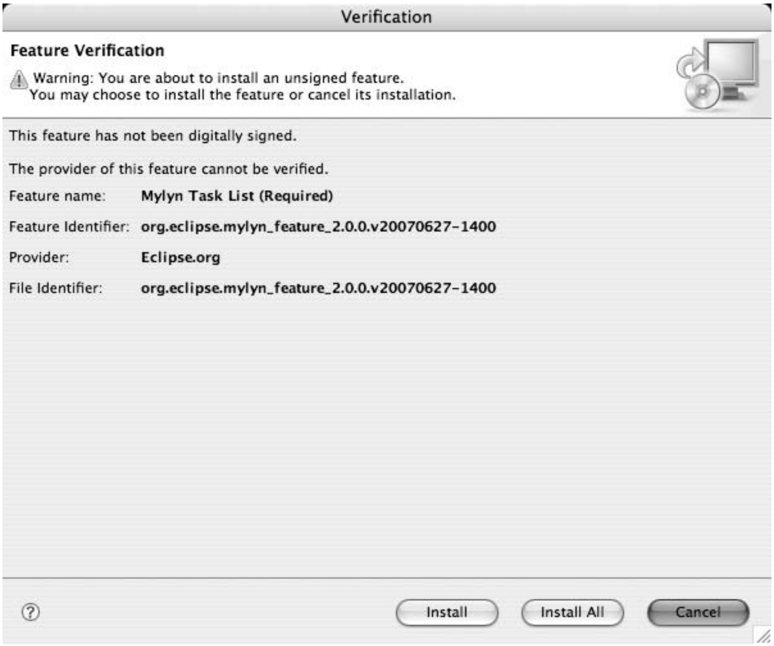


Figure 2-12. *Verifying the unsigned plug-in*

This screen is called Feature Verification, but it's really complaining about the Mylyn package being unsigned. You should get used to this. While cryptographic signing of features is a neat idea, it doesn't happen much. Just click Install All.

A few windows will flit by as Mylyn is installed, and once it's complete, Eclipse will ask if you want to apply the changes or restart (see Figure 2-13).



Figure 2-13. *Installation complete*

Choose Yes to restart Eclipse. When it restarts, it will go to the Eclipse overview, which is shown in Figure 2-14. In the middle of the top bar is the folder over arrow, which will take you back to the workspace. Go there and rejoice in your new accomplishment. Your Eclipse installation has just grown a new capability, even if you haven't used it yet.

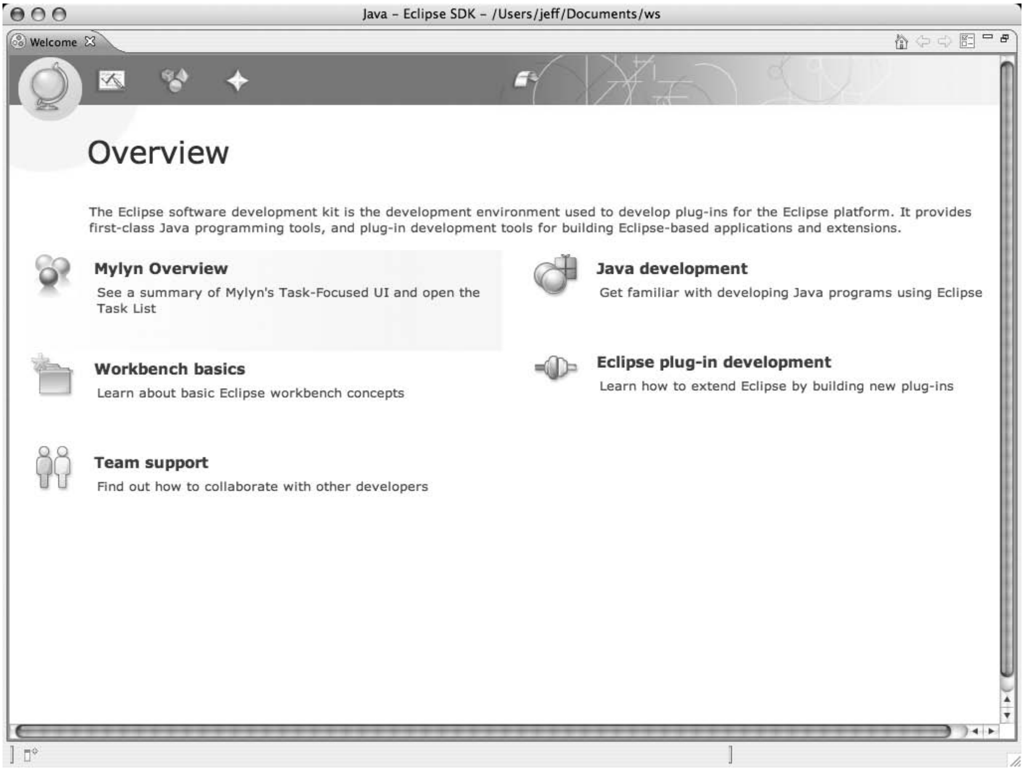


Figure 2-14. *The Overview screen after the installation has completed*

Installing and Configuring Pydev

Mylyn is installed once Eclipse restarts. Installing Pydev is the next step. The Pydev web site is `http://pydev.sourceforge.net/`. The Pydev update site is `http://pydev.sourceforge.net/updates/`. Follow the same procedure as with Mylyn.

Once Pydev is installed, it must be configured. Out of the box, it doesn't know where Python is located, so the first step is configuring the Python interpreter. If you're on Linux, then the Python interpreter will probably be located in `/usr/bin/python`; if you're on OS X, it will probably be located in `/Library/Frameworks/Python.framework/Versions/2.5/bin/python`; and if you're on Windows, it will probably be in `C:\Python25`.

There are three steps. First, open Window ➤ Preferences ➤ Interpreter ➤ Python. Second, enter the path you previously chose. Third, choose the paths to be in your System Python path. You should *not* select folders that will be used in your project, but when starting out, that shouldn't be a problem. By default, it only checks the correct paths, so you'll only need to worry when you start doing more complicated things.

■ **Note** The Python path is a series of directories that are searched when packages are imported. The `PYTHONPATH` variable contains additional directories that are searched.

When you click OK, it should process through the libraries very quickly, and you should find yourself back at the Python Interpreters screen (shown in Figure 2-15).

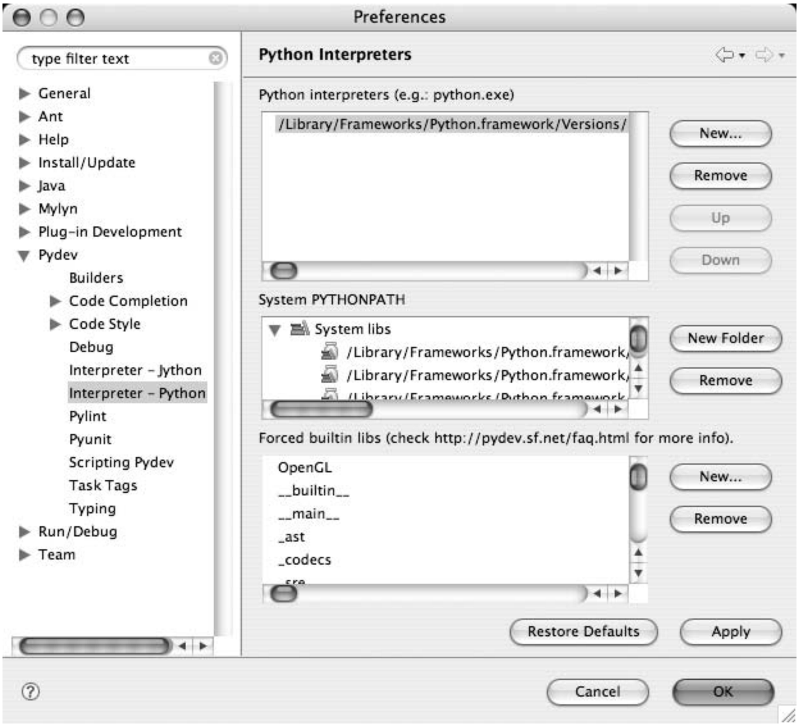


Figure 2-15. *The Python interpreters have been set.*

The system `PYTHONPATH` box has been filled in with the values from the browser, and so has the "Forced builtin libs" box. You should see "__builtin__" and a slew of other libraries with underscore-prefix names.

When you click OK, you'll see a window entitled Progress Information, and you'll watch a thousand or two module names go flying by as Pydev builds its cache.

Your First Project

At this point, you can start working on a project. Choose File ➤ New ➤ Project. From there, you can select Pydev ➤ Pydev Project. You should see the window shown in Figure 2-16.

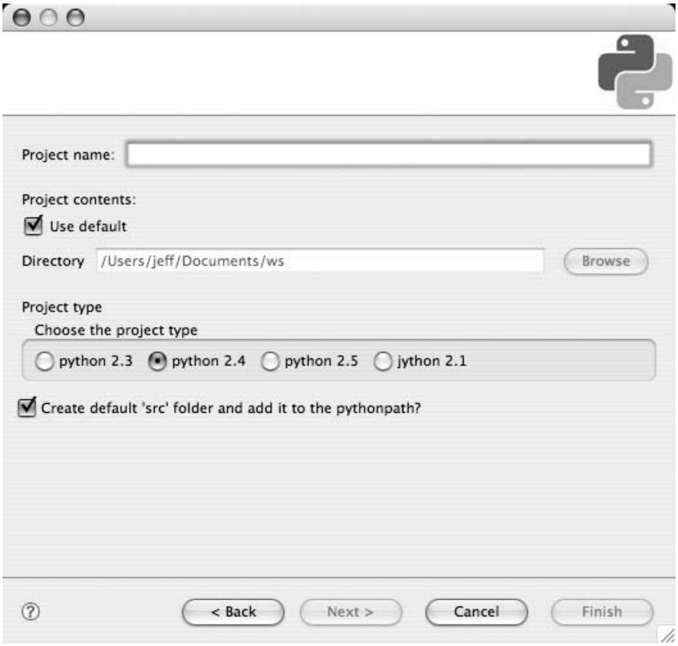


Figure 2-16. *Starting a new project*

Enter **agile** in the "Project name" field. Choose "python 2.5" from the list of project types. (For those of you wondering about the "jython 2.1" option, Jython is a Java-based Python interpreter. I'm not using it in this book.) The final option, labeled "Create default 'src' folder and add it to the pythonpath?" should remain checked.

Source folders are Eclipse directories that contain code. They are automatically added to the Python interpreter's path. To do any development, you need at least one in your project, but it is possible to leave this box unchecked. If you do, then you'll have to add the directories later. You could click Next at this point to reference code in other modules, but we're not doing that in our current project, so click Finish, which will return you to the workbench, as shown in Figure 2-17.

On the left-hand side in a pane entitled Pydev Package Explorer, you'll now see a blue folder entitled `agile`. If you open it, you'll see the `src` folder inside. (You should also note that, up in the right-hand corner, the active icon in the perspective tab shows you are in the Pydev perspective.)

Now you've done the grunt work of setting up a project, and you're at the point where you can start working with Python. You're going to create a Python class called `examples.hello.world.Greet`. Pydev calls library directories *packages*, and it calls source files *modules*. You'll create the package structure, and then create the module.

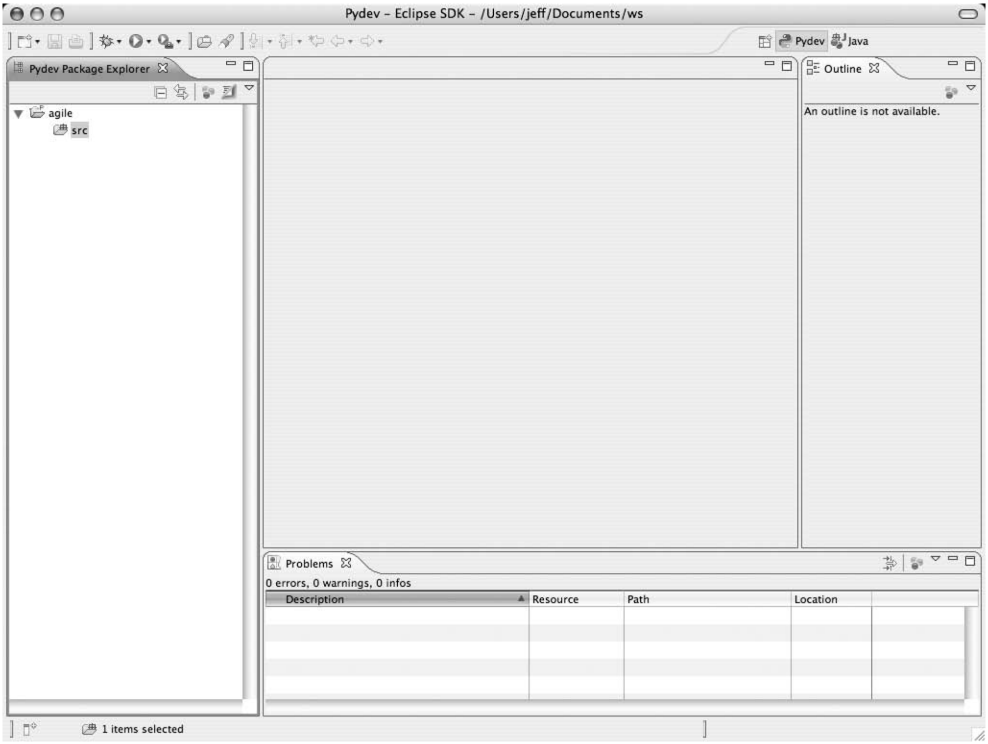


Figure 2-17. *The workbench with your new project*

Right-click the `src` directory and select New ➤ Pydev Package. This brings up a window with two fields, as shown in Figure 2-18.

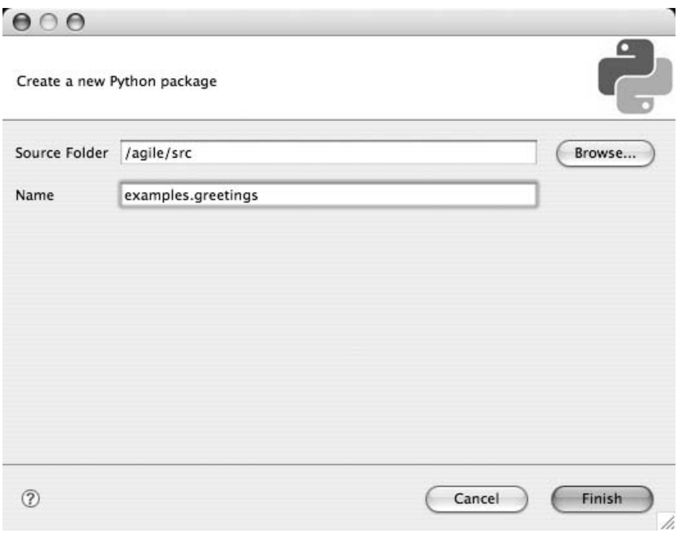


Figure 2-18. *Creating a new project*

The two fields are Source Folder and Name. Source Folder is already filled in with /agile/src, and Name is empty; enter the package name examples.greetings as shown previously, and then click Finish. This creates the named packages and all of the __init__.py files, and takes you back to the workbench, as shown in Figure 2-19.

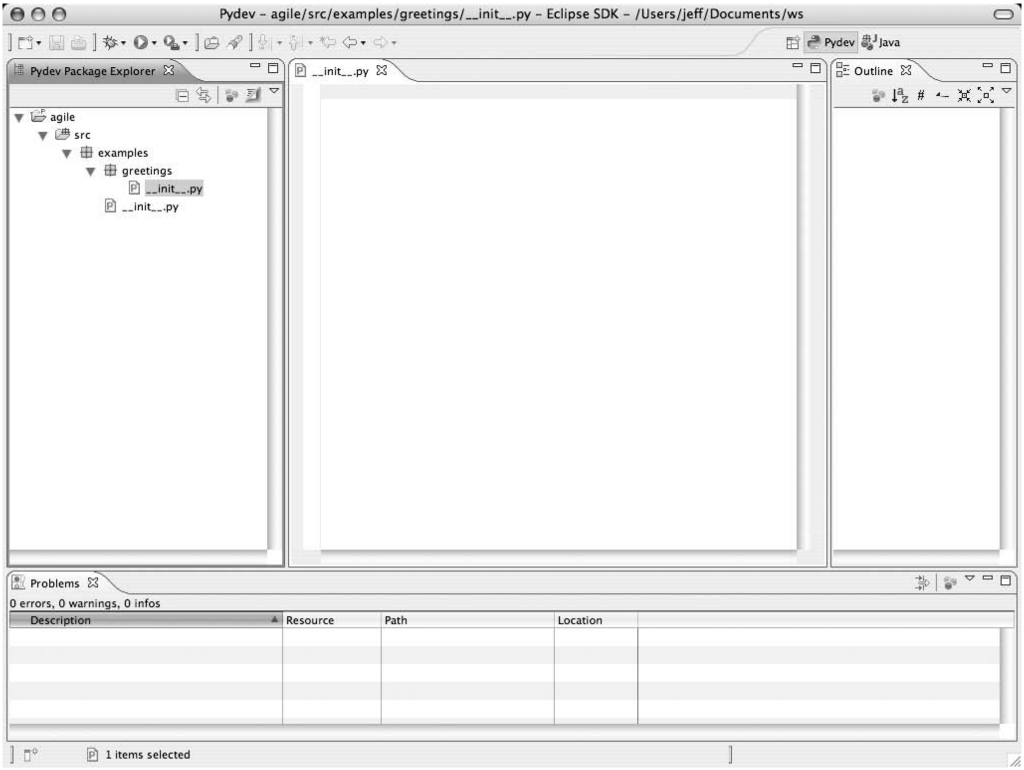


Figure 2-19. *The new packages have been created.*

Now you'll create the module examples.greetings.standard. Right-click the greetings package in the Package Explorer on the left side of the workspace. Select New ➤ Pydev Module. That will bring up the (unnamed) module creation window, shown in Figure 2-20.



Figure 2-20. *Choosing the module name*

The window has three fields: Source Folder, Package, and Name. The first two are filled in for you. Enter **standard** into the Name field, and then click Finish. You will be taken back to the workbench, which should look something like Figure 2-21.

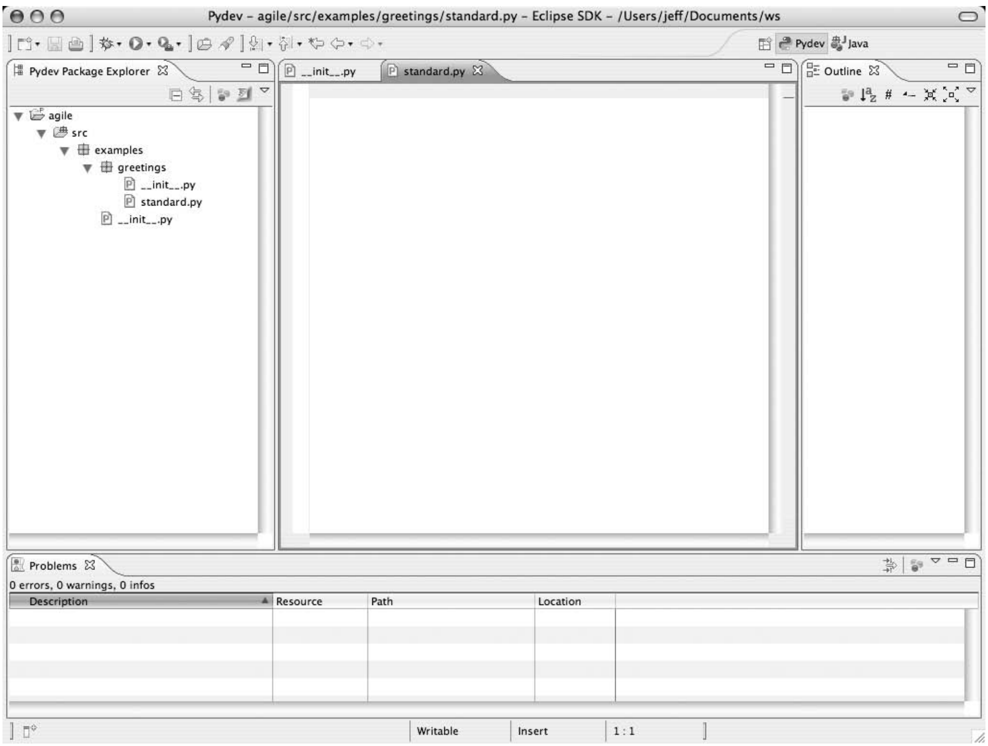


Figure 2-21. *The module has been created.*

You'll notice that `standard.py` shows up in the Package Explorer on the left, and that the editor in the center pane is open to this file. Click into that window and enter the following program:

```
#!/usr/bin/python

class HelloWorld(object):

    def main(self):
        print "Hello World!"

if __name__ == '__main__':
    HelloWorld().main()
```

You'll notice several things while you type, the most interesting of which is that Pydev makes guesses about what you're about to type. In the case of the `def main`, it makes the correct guess, but it doesn't move the cursor. You can either continue typing, or you can accept Pydev's guess by pressing Ctrl+Enter. After you enter and save this text, the workspace will look something like Figure 2-22.

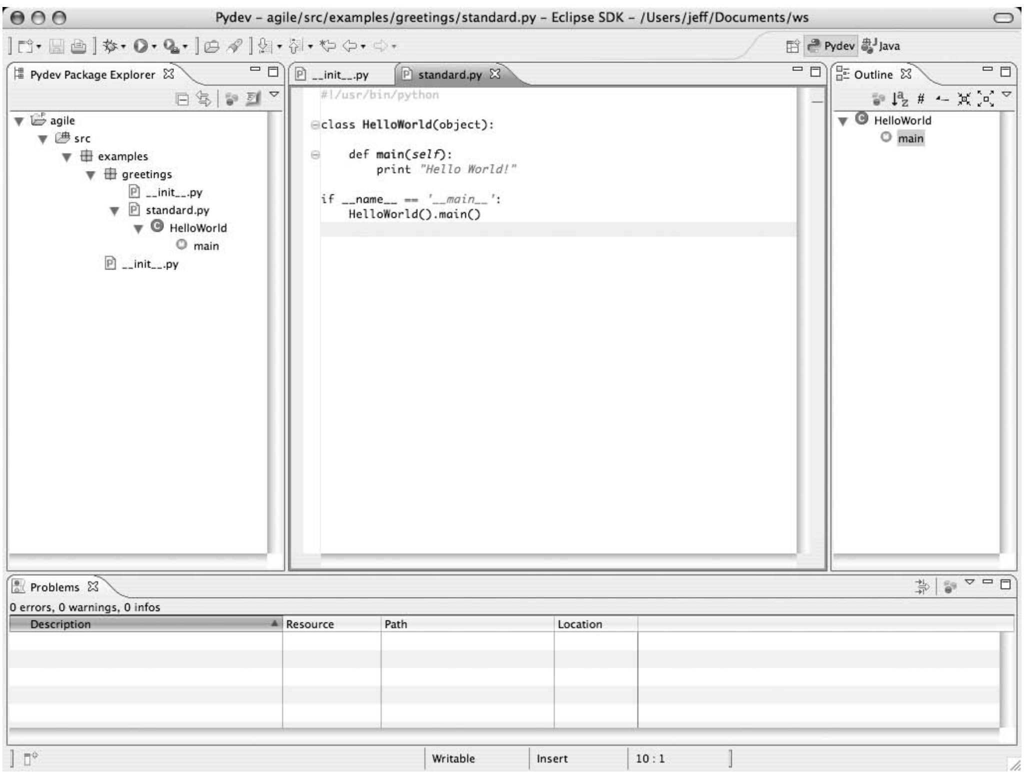


Figure 2-22. *The HelloWorld class has been created.*

As you typed, Pydev also updated the Package Explorer on the left and the Outline view on the right. You can use both to navigate through your program. Double-clicking any node in the Package Explorer opens an editor for the corresponding file. The cursor is positioned at the point in the program that the node in the explorer represents. The Outline view to the right shows the elements corresponding to the currently active editor. Clicking any element in the Outline view will also take you to the corresponding definition.

Running the program is easy. Right-click inside the text editor, or from the context menu select Run As ➤ 4 Python Run. The program's output will appear in a Console view at the bottom of the workspace. You can also bring up the same menu by right-clicking the `standard.py` module in the Package Explorer window.

Looking Under the Hood

Many external tools interact with the code stored in Eclipse. In order to work with them, it is necessary to understand how Eclipse lays out its directories. To begin, you'll need to change directories to the root of your Eclipse workspace; on my machine, it is `~/Documents/workspace`. From there, we'll go on a brief tour. (My machine's name is phytoplankton and my username is jeff.)

```
phytoplankton:~ jeff$ cd ~/Documents/workspace
phytoplankton:~/Documents/workspace jeff$ ls -aF
```

```
. ./              ../.              .metadata       agile/
```

The directory `.metadata` contains all of your Eclipse preferences and system-wide configuration data. We really don't care much about the details of this directory. The directory `agile` is of more interest. It contains the project that we are working with, so we'll take a look inside there.

```
phytoplankton:~/Documents/workspace jeff$ cd agile
phytoplankton:~/Documents/workspace/agile jeff$ ls -aF
```

```
./              ../             .project        .pydevproject   src/
```

Besides the ubiquitous `.` and `..`, there are three filesystem entries here: two files and one directory. The file `.project` contains the Eclipse configuration information for the project. The file `.pydevproject` contains the project configuration information for Pydev. Both are XML files, and they are specific to the project.

■ **Note** Under the UNIX filesystem, every directory has two subdirectories. The directory named `.` (one period) is the current directory. The directory named `..` is the parent directory. These are shown, but they are ignored during this discussion.

The directory src is the source folder you defined when the project was created. It contains the Python packages and modules that you defined. They're just normal Python library directories with __init__.py files and .py files.

```
phytoplankton:~/Documents/workspace/agile jeff$ cd src
phytoplankton:~/Documents/workspace/agile/src jeff$ ls -aF
```

./	../	examples/

```
phytoplankton:~/Documents/workspace/agile/src jeff$ ls -aF examples
```

./	../	__init__.py	greetings/

```
phytoplankton:~/Documents/workspace/agile/src jeff$ ls -aF examples/greetings
```

./	../	__init__.py	standard.py

These are just normal Python files organized just like you'd expect. If you set the PYTHONPATH to the root of the source directory (~/Documents/workspace/agile/src) you can develop from the command line using the same files. In fact you're going to be doing that often in the coming chapters.

Paying for More Functionality

At this point, I'd suggest trying out Pydev Extensions. If you end up using Eclipse for Python development (as I hope you do), then it is a worthwhile investment. It gives you a number of features missing from the free version of Pydev, most of them relating to much-improved code analysis. Notably, it includes a much better "go to definition of function" feature for navigating your code.

Pydev uses an external program (Bicycle Repair Man) that has trouble with class methods, but Pydev Extensions improves upon this. It does real-time code analysis, and its results are superior to those of vanilla Pydev. It offers auto-management of imports and much better code completion. It offers a wider variety of refactorings and the ability to do remote debugging. I could go on, but I'll let you explore on your own.

The extensions have a 30-day free trial. They continue working after the trial is over, but they nag you every couple of hours, suggesting that you should buy a copy. The nagging isn't a hindrance to your work, but it is sufficiently annoying that you'll probably give in and buy the software just to make the messages go away. The web site for Pydev Extensions is http://fabioz.com/pydev/. The download site is www.fabioz.com/pydev/updates/. The extensions install from the update site, just as with Mylyn and Pydev.

Summary

Eclipse is an increasingly popular IDE that offers many advantages over the command line. It's free, and it has an open plug-in architecture that is being exploited by many component producers. There are plug-ins available for a plethora of purposes, many of which are relevant to this book.

Plug-ins are easily loaded from within Eclipse itself. You've done this with several so far. Mylyn is the most detailed example of the procedure, but more significantly we've installed the Pydev plug-in and configured it.

Pydev turns Eclipse into a Python development environment. It acts as a sophisticated wrapper around the system Python installations. It includes everything users have come to expect from an IDE, providing structural browsers, integrated editors, and code completion. It has a number of features that haven't been examined, as well. These include unit testing support, an integrated graphical debugger, remote debugging, and refactoring support.

There is a slew of additional features that are added by Pydev Extensions. It's free for 30 days, so it's worth looking at. If you decide the features aren't worth the money, then you can turn Pydev Extensions off again through the plug-in management screens.

Now that I've introduced a basic working environment, I'll examine revision control and source repositories. Revision control is the technology that underlies the agile practice of continuous integration. The source repository is where file revisions are stored, and it is the location where all the group's development work resides. It is the means by which members' changes are integrated on a daily basis, and it serves as the source of all truth for the code base. Users get the most recent code from here, and so does all of the build automation.

Subversion is the revision control system examined in this book. It's free, and it's in common use. It's a favorite with developers, and it is far superior to its predecessor, CVS. It can be run locally or as a remote service. In the next chapter you'll learn how to use it both from the command line and from within Eclipse using the Subversive plug-in.

CHAPTER 3

Revision Control: Subverting Your Code

At one point or another in your days as a developer, chances are you've mistakenly deleted an important file. It happens. Sometimes the problem is worse than you thought; for instance, you might not even be sure what files you deleted. Or perhaps your brilliant idea for a new feature has horribly broken the code base. Making matters worse, your changes were spread across multiple files, and now it's unclear how to return to the previous state. Version control can remedy all of these problems by coordinating the life cycle of all files in your project, allowing you to not only recover mistakenly deleted code, but actually revert back to earlier versions.

Version control can go well beyond simple file management and recovery, though; it also plays a crucial role in managing changes made in environments where multiple developers might be simultaneously working with the code. Sure, each of you could make copies of the code base and yell over the cubicle wall, "Hey, I'm working on `tools.py` right now, don't touch it." But sooner or later, you'll nonetheless overwrite each other's changes. It gets worse when you're not within earshot, or even the same time zone.

Revision control helps this situation by acting as a moderator and a single source of truth. Either by gating access or merging changes, it prevents you from stepping on each other's toes. Revision control keeps track of what changes were made, and further, it keeps track of who made them.

Revision control also lets you work on multiple versions of the code at the same time, allowing you to test out that ambitious new feature without interfering with the stable version. This encourages all sorts of efficiencies, allowing one developer to add new features for an upcoming release while another developer works on security fixes for the current release. When you are ready, the changes can be merged back together.

The benefits of coordination aren't limited to humans, though. You can configure your build process to execute against the source repository and cause the build to begin anew any time somebody checks in new code.

You can also use revision control to enforce policy. For instance, you can prevent users from checking in changes to certain branches of the tree, analyze code before allowing it to be submitted, ensure that all Python code has proper whitespacing, or require that all Python files are syntactically correct. All of this is made possible by revision control.

Subversion is one of the most widely used revision control systems available. In this chapter, I'll show you how to use Subversion to manage your code on your local machine, both from the command line and from within Eclipse via the Subversive plug-in. The examples

include such common operations as adding, editing, and removing files, but they also include operations that don't immediately spring to mind. Among these are comparing your local changes with those in the revision system, retrieving others' work from the repository, and resolving conflicts between changes you have made and changes that others have made.

Revision Control Phylum

We can look at revision control systems in a couple of broad aspects. The most significant of these is distributed vs. centralized. Another is availability. Is the repository available locally or remotely? I'm not even going to mention revision control systems that are local. Many of the practices in this book are intended to scale up to multiple machines, so a local repository just doesn't work for us.

Centralized revision control systems have been around forever. They access a single logical repository that is physically stored on one or more systems. Most commercial systems are centralized, and centralized systems seem to be the most mature. Examples of centralized revision control systems are CVS, Visual SourceSafe, Subversion, Perforce, and ClearCase.

Distributed revision control systems are the new kid on the block. To date, their most highly visible implementations have been related to operating system kernel development. Both Linux and Solaris use the distributed repository Git, which was created to support development of Linux. Examples of distributed revision control systems are Darcs (`darcs.net/`), BitKeeper (`www.bitkeeper.com/`), Mercurial (`www.selenic.com/mercurial/wiki/`), Git (`git.or.cz/`), and Bazaar (`bazaar-vcs.org/`). They're pretty cool in some conceptual ways, but many release engineering professionals look on them warily. Despite their complexities, kernels are still simple and well-understood entities compared to many enterprise systems, and distributed version control systems have yet to prove themselves in the more complicated enterprise environments.

If you look on the Web, you'll see vociferous arguments about which kind of revision control system is better. Much of this seems to be driven by people's experience with CVS. Advantages are touted for distributed revision control that when examined closely boil down to "Our software doesn't suck like CVS." Claims are made about branch creation, labeling, or merging that boil down to "CVS did it like this. CVS is a centralized revision control system. Therefore, all centralized revision control systems do it like this." Examples of where this logic is applied include branching and merging. Almost never are the free systems compared to the commercial systems.

The commercial systems are impressive. In general, they're more mature and feature rich than the free systems. They offer administrative controls and reporting that is missing from the free systems. They do branching and merging well, too. Perforce particularly shines in this area, and its integration tools are impressive. However, we're not going to be using Perforce in this book.

We'll be using Subversion. The choice is driven by a number of factors. First, Subversion is widely used. As with Eclipse, there is a large ecology of tools associated with it. The tools we've worked with and will be working with later easily integrate with it. And it's free.

Subversion supports atomic commits of multiple files. There is a global revision counter allowing you step back to any specific point in the repository's history. It supports labeling and branching. If some of these terms don't make sense to you right now, they will shortly.

What Subversion Does for You

Subversion stores your code on a central server in a repository. The repository acts much like a filesystem. Clients can connect to the filesystem and read and write files. What makes the repository special is that every change ever made to any file in the repository is available. Even information such as renaming files or directories is tracked.

Clients aren't limited to looking at the most recent changes. They can ask for specific revisions of a file, or information like, "who made the third change last Thursday?" This is where the real utility of a revision control system comes from.

A user checks out code from the repository, makes changes of one sort or another, and then submits those changes back to the repository. Multiple users can be doing this at the same time. Two or more users can check out the same file and edit it, and when the file changes are submitted, they'll have to resolve any conflicts. This resolution is called *merging*.

The overall process is called *edit-and-merge*. Contrast this with the other approach, called *exclusive locking*. In this scheme, only one person gets to have a file open for edit at any time. While it saves the possible work of merging changes, it can bring development to a halt. It turns out that in practice, edit-and-merge is the least disruptive.

What happens if two users try to submit changes at the same time? One goes first. In Subversion, groups of files are submitted together. The submissions are a single atomic action. While CVS has interfaces that allow you to submit multiple files at once, each file is an individual submit. Two users can submit sets of files, and their changes will be interleaved. This can never happen with Subversion.

Subversion maintains a global revision counter that is incremented with every submission. It increases monotonically, and it can be thought of as describing the state of the repository at any point in time. While it may not seem like much, having this counter is remarkably useful for labeling builds and releases.

Subversion stores working copies of the files on your disk. It stores the the information describing these working copies on your local system too. This contrasts with other systems that store this state on a server. Subversion doesn't need to contact the server to find out the current state of your files, allowing you to work remotely without a network connection. The bad news is that you must be connected to rename or copy files, which takes away from the joy.

The local state is stored in directories named `.svn` (just like CVS uses `.cvs` directories). There is one in every directory checked out from Subversion. Many refer to these directories as "droppings." The `.svn` directories carry virgin copies of all files in your working copy. This way, the more frequently invoked commands, such as diff and revert, can be run without accessing the central repository.

Frequently, there is a need to work on multiple differing copies of a project. Consider a software product that has an installed base of users. At most points in a software product's life, there will be multiple activities going on. Some developers will be working on new features for upcoming releases. Other developers will be working on high-priority repairs for customers who have already installed the product. The new features will destabilize the codeline and often mask the bugs that are reported by customers. They'll also introduce many new bugs, particularly early in the development cycle. High-priority bug fixes must be made to code that mirrors the release code as closely as possible so that the customer doesn't receive a version of the product that is broken in yet more new and interesting ways.

Sadly, both the new development and bug fixes must be performed simultaneously. This is done by creating copies of the program. One copy is used for the new work, and the other is used only for the bug fixes. These copies are referred to as *branches*. Branches are independent but related copies of a program. A new branch can be made whenever simultaneous but conflicting changes must be made to a program.

In practice, managing branching is one of the primary jobs of a revision control system. As branches proliferate, it is necessary to have some way of referring to them. This is done with *labels*. Labels are names attached to branches at a particular point in time. They let you precisely and concisely specify a version of a program.

The new release will require the bug fixes from the maintenance branch, so the branches will need to be recombined. This process is called *merging*. This is an important part of branch management. Merging takes the changes from one branch and combines them with another branch. A surprisingly large part of the process can be automated, and the results work a surprisingly large percentage of the time, but ensuring that they work requires good tests, and the process almost always requires some developer intervention.

Subversion supports branching—that's the good news. The bad news is that merging support is very new. It was just added in version 1.5, and it has yet to be widely deployed.

That brings us to labeling. Subversion supports labeling. Kinda. Labeling is just branching to a different place. The good news is that we have the global revision counter, which allows us to bypass labeling to some degree.

Getting Subverted

The first step is installing Subversion. Subversion is available from `http://subversion.tigris.org/`. If you're running on Linux and you installed your system with development tools included, then the odds are good that you've already got Subversion installed. If Subversion is not installed, chances are that packaged binaries can be located for your system at `http://subversion.tigris.org/project_packages.html`, and if worse comes to worst, the source code is also available there.

Once Subversion is installed, the first step in creating your repository is initializing the database on your Subversion server:

```
phytoplankton:~ jeff$ svnadmin create /usr/local/svn/repos
```

This creates the Subversion database in the directory `/usr/local/svn/repos`. There are two ways of storing this information. One is on the filesystem, and the other is in Berkeley DB database files. The default is within the filesystem, and unless you have good reason to do otherwise, I suggest taking the default. You can find more information in *Practical Subversion, Second Edition*, by Daniel Berlin and Garret Rooney (Apress, 2006). The directory structure that will be created looks something the following:

```
$ ls -F /usr/local/svn/repos
```

```
README.txt      dav/            format          locks/
conf/           db/             hooks/
```

■ **Note** You may need to create the directory `/usr/local/svn` before you can run this command, and you may also need to set your permissions appropriately. I had to change ownership to my own account. If I were running this in production, it would be owned by the `svn` user.

Subversion repositories can be accessed in multiple ways. The path to and within the repository is specified using a URL (see Figure 3-1). The URL scheme (the part before the first colon) specifies the access protocol. This can be through the local filesystem, HTTP or HTTPS, SSH, or Subversion's own protocol.

The scheme you use will depend on the server that you're accessing. The easiest is the file protocol. It can only be used when you're on the same machine as the Subversion server. The HTTP and HTTPS protocols require the use of Apache. You gain a huge amount of flexibility in access control by using Apache, but the setup is more complex. The Subversion protocol is somewhere in between. It uses a dedicated server that is very easy to set up, and it offers some level of access control. The protocol is faster than using HTTP or HTTPS for large projects. We're going to be using the file protocol for the examples in this section of the book.

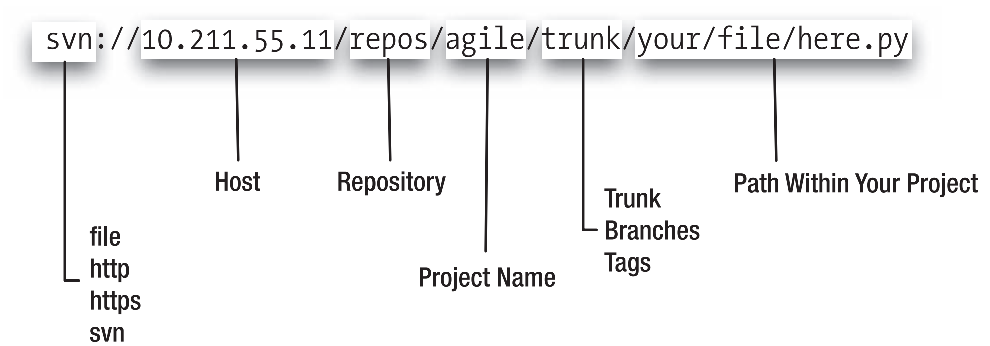


Figure 3-1. *Parts of a Subversion URL*

The process of loading a project into Subversion involves several steps. The first is the creation of a repository, which you've already done. A repository can hold any number of projects, and these projects can be organized in any number of ways. You have to decide how you're going to do that. Then you have to create those directories, and finally you'll be able to import the project into Subversion.

In most working environments, there are multiple projects within a single repository. This requires some level of organization. Generally, these projects have a mainline containing the gold version of the code. They have a number of branches where conflicting work is performed, and they have a place for tags. (*Tag* is Subversion's term for a label.) By convention, the main codeline is stored in a directory called `trunk`, branches are stored in a directory named `branches`, and tags are stored in a directory named `tags`. We'll stick with that convention.

There are two common conventions for organizing projects. One is *project major*, and the other is *project minor*. In project major, each project has its own `trunk`, `branches`, and `tags` directories. In project minor, the repository has top-level `trunk`, `branches`, and `tags` directories. Beneath each of these is a directory for each project, as shown in Figure 3-2.

Project Major Organization	Project Minor Organization
/path/to/repository	/path/to/repository
/path/to/repository/project1	/path/to/repository/trunk
/path/to/repository/project1/trunk	/path/to/repository/trunk/project1
/path/to/repository/project1/branches	/path/to/repository/trunk/project2
/path/to/repository/project1/tags	/path/to/repository/branches
/path/to/repository/project2	/path/to/repository/branches/project1
/path/to/repository/project2/trunk	/path/to/repository/branches/project2
/path/to/repository/project2/branches	/path/to/repository/tags
/path/to/repository/project2/tags	/path/to/repository/tags/project1
	/path/to/repository/tags/project2

Figure 3-2. *Project major and project minor organization*

I prefer project major organization. It makes it easy to identify what belongs to a project, it makes access control easier to manage, and it allows you to move your project about with very few commands. Our project is named agile. With project major organization, our directories structure will look like this:

```
/usr/local/svn/repos/agile
/usr/local/svn/repos/agile/trunk
/usr/local/svn/repos/agile/branches
/usr/local/svn/repos/agile/tags
```

You create this with the command `svn mkdir`. Once you've created the directories, you can look at them with the `svn list` command:

```
phytoplankton:~ jeff$ svn mkdir file:///usr/local/svn/repos/agile \
  -m "creating the internal organization for the project 'agile'"
```

```
Committed revision 1.
```

```
phytoplankton:~ jeff$ svn list file:///usr/local/svn/repos/agile
phytoplankton:~ jeff$ svn mkdir \ file:///usr/local/svn/repos/agile/trunk ➥
file:///usr/local/svn/repos/agile/branches \ file:///usr/local/svn/repos/agile➥
/tags -m "creating the internal organization for the project 'agile'"
```

```
Committed revision 2.
```

```
phytoplankton:~ jeff$ svn list file:///usr/local/svn/repos/agile
```

```
branches/
tags/
trunk/
```

Now you can import the mainline into the depot. This is done with the `import` command. The `import` command takes three arguments:

- `Agile` is the imported directory.
- The `file:` URL is the destination in `trunk`.
- The `-m` option is the `commit` comment.

The contents of the directory `agile` will be loaded into the Subversion trunk. The directory `agile` itself will be omitted:

```
phytoplankton:~ jeff$ cd ~/ws
phytoplankton:~/ws jeff$ svn import agile \
 file:///usr/local/svn/repos/agile/trunk \
 -m "Initial import of our the 'agile' trunk"
```

```
Adding         agile/.project
Adding         agile/src
Adding         agile/src/examples
Adding         agile/src/examples/__init__.py
Adding         agile/src/examples/greetings
Adding         agile/src/examples/greetings/__init__.py
Adding         agile/src/examples/greetings/standard.py
Adding         agile/.pydevproject

Committed revision 3.
```

```
phytoplankton:~/ws jeff$ svn list \ file:///usr/local/svn/repos/agile/trunk
```

```
.project
.pydevproject
src/
```

You have imported the `.project` and `.pydevproject` files that Eclipse created. These files are as important as any other source files. As you create larger and more complicated projects, these files will contain more and more information that you don't want to lose. When a developer checks out a file from Subversion the first time, they will be able to import it directly into Eclipse. They'll be working on the code rather than figuring out how get the code to build under Eclipse.

Working with Your Subverted Code

At this point, you've imported your code into Subversion, but you don't have a working version on your local machine. You can't add new files, edit files, delete files, or update from the repository until you get a local copy.

Your local copy can't be pulled directly into your workspace directory. Subversion will detect that the files already exist. You need to do one of two things: either move aside your current project directory or pull the code down into your existing directory. In this case, choosing a new project is what you'll do next:

```
phytoplankton:~/ws jeff$ svn checkout \ file:///usr/local/svn/repos/agile/trunk ➥
hello
```

```
A    hello/.project
A    hello/src
A    hello/src/examples
A    hello/src/examples/__init__.py
A    hello/src/examples/greetings
A    hello/src/examples/greetings/__init__.py
A    hello/src/examples/greetings/standard.py
A    hello/.pydevproject
Checked out revision 3.
```

```
phytoplankton:~/ws jeff$ ls -la hello
```

```
total 16
drwxr-xr-x   6 jeff  jeff  204 Oct  2 18:51 .
drwxr-xr-x   5 jeff  jeff  170 Oct  2 18:51 ..
-rw-r--r--   1 jeff  jeff  359 Oct  2 18:51 .project
-rw-r--r--   1 jeff  jeff  307 Oct  2 18:51 .pydevproject
drwxr-xr-x   8 jeff  jeff  272 Oct  2 18:51 .svn
drwxr-xr-x   4 jeff  jeff  136 Oct  2 18:51 src
```

The first thing to notice is the `.svn` directory. Each directory checked out from Subversion will contain one. This is where Subversion stores information describing the state of your local system. It contains a record of each file that has been checked out and a copy of that file.

You've already seen how to perform a few common operations. You've made directories in the repository; you've listed the contents of a directory; and you've looked at the contents of a file. I'll run through the rest of the operations you'll routinely perform with Subversion. These are the operations that every user needs. They include the following:

- Examining your working copy
- Adding a file
- Deleting a file
- Reverting a file
- Committing changes
- Editing a file
- Comparing a file against the repository
- Updating your working copy
- Resolving conflicts during a submission

These operations form the core of what you'll be doing from day to day. They will carry over almost directly to the Eclipse interface. We'll start by examining the files.

Examining Files

There are two commands that are used to examine the state of your workspace. They are `svn info` and `svn status`. `svn info` works on individual files and directories. `svn status` works on your workspace as a whole. `svn status` is used more frequently than `svn info`, but there are times when you need information that is only available through `svn info`, so we'll start there.

```
phytoplankton:~/ws jeff$ cd hello
phytoplankton:~/ws/hello jeff$ svn info
```

```
Path: .
URL: file:///usr/local/svn/repos/agile/hello
Repository Root: file:///usr/local/svn/repos
Repository UUID: 74a71bd7-8c3b-0410-b727-f8ad94e0a8f0
Revision: 3
Node Kind: directory
Schedule: normal
Last Changed Author: jeff
Last Changed Rev: 3
Last Changed Date: 2007-10-02 18:46:37 -0700 (Tue, 02 Oct 2007)
```

```
phytoplankton:~/ws/hello jeff$ svn info .project
```

```
Path: .project
Name: .project
URL: file:///usr/local/svn/repos/agile/hello/.project
Repository Root: file:///usr/local/svn/repos
Repository UUID: 74a71bd7-8c3b-0410-b727-f8ad94e0a8f0
Revision: 3
Node Kind: file
Schedule: normal
Last Changed Author: jeff
Last Changed Rev: 3
Last Changed Date: 2007-10-02 18:46:37 -0700 (Tue, 02 Oct 2007)
Text Last Updated: 2007-10-02 18:51:35 -0700 (Tue, 02 Oct 2007)
Checksum: 97703150e87f434355444a9f07b6750b
```

Notice that Subversion tracks the directory itself. This is reported in the `Node Kind` field. This differs from some other version control systems that only track files. The really important field here is `Revision`. It lets you know what edition of a file the system thinks you have. You can get this information for all files using the `svn status` command.

Run without arguments, `svn status` reports changed files that have not been committed. You have no changed files at this moment, so it would report nothing. You're interested in seeing the verbose output, which shows all files. You turn this on with the `-v` flag:

```
phytoplankton:~/ws/hello jeff$ svn status -v
```

```
                3       3 jeff         .
                3       3 jeff         .project
                3       3 jeff         src
                3       3 jeff         src/examples
                3       3 jeff         src/examples/__init__.py
                3       3 jeff         src/examples/greetings
                3       3 jeff         src/examples/.../__init__.py
                3       3 jeff         src/examples/.../standard.py
                3       3 jeff         .pydevproject
```

You can't tell easily, but there are a number of blank fields ahead of the first numbers. The four remaining fields are the working revision, the head revision, the author committing that head revision, and finally the path to the file. This information will become more interesting as you work. Now that you know how to look at your workspace, you can move on to making changes.

Adding Files

Suppose that you've created a new file named `src/examples/common.py`, and you want to add this file to the repository. You do this with the `svn add` command. It works pretty much as you'd expect. We'll look at its effects using the `svn status` command:

```
phytoplankton:~/ws/hello jeff$ svn add src/examples/common.py
```

```
A         src/examples/common.py
```

```
phytoplankton:~/ws/hello jeff$ svn status
```

```
A      src/examples/common.py
```

```
phytoplankton:~/ws/hello jeff$ svn status -v
```

```
...
                3       3 jeff         src/examples
A               0       ?   ?          src/examples/common.py
                3       3 jeff         src/examples/__init__.py
...
```

Notice that `status -v` shows an `A`, which denotes a file to be added. It shows that the current revision is `0`, which denotes that there's no revision on the client and that the head revision and head author don't exist. This demonstrates something important about Subversion. Adding a file doesn't immediately add the file to the repository. It adds it to the list of pending changes. In SVN parlance, this is known as *scheduling an add for commit*. You have to use `svn commit` to complete the addition.

```
phytoplankton:~/ws/hello jeff$ svn commit -m "Adding common code for all greetings"
```

```
Adding         src/examples/common.py
Transmitting file data .
Committed revision 4.
```

```
phytoplankton:~/ws/hello jeff$ svn status -v
```

```
...
                3        3 jeff         src/examples
                4        4 jeff         src/examples/common.py
                3        3 jeff         src/examples/__init__.py
...
```

Now that the change is committed, you can see that the file has been added to the repository. The file was committed in revision 4, and you have that revision in your working copy.

Copying and Moving Files

Unlike several other revision control systems, Subversion has simple commands for copying and moving files. These commands maintain revision history and ancestry between the source and destinations. We'll copy `common.py` to `shared.py`:

```
$ cd src/examples
$ svn copy common.py shared.py
```

```
A         shared.py
```

```
$ svn status
```

```
A  +   shared.py
```

```
$ svn commit -m "Copying common.py to shared.py"
```

```
Adding         examples/shared.py

Committed revision 5.
```

You'll notice that `svn status` returns `A +`. The + indicates that revision history is being maintained from the original to the copy. A similar process happens with a move:

```
$ svn move shared.py unshared.py
```

```
A         unshared.py
D         shared.py
```

```
$ svn status
```

```
A  +   unshared.py
D      shared.py
```

```
$ svn commit -m "Moving shared.py to unshared.py"
```

```
Deleting       examples/shared.py
Adding         examples/unshared.py

Committed revision 6.
```

In this case, there are two changes that are performed. The line beginning with `A +` indicates that `unshared.py` was added while maintaining history, and the line beginning with `D` indicates that the original file `shared.py` was deleted.

This is also the first time you've seen multiple changes at once. Unlike CVS, both of these changes are performed in a single atomic transaction. At no point is there a moment where both files exist. To the outside world, it is as if the copy and delete happened simultaneously.

Deleting Files

The `svn delete` command schedules files for removal. The `svn status` command shows these prefixed with `D`. These changes become permanent when they are committed.

```
$ svn delete common.py unshared.py
```

```
D         common.py
D         unshared.py
```

```
$ svn status
```

```
D      common.py
D      unshared.py
```

```
$ svn commit -m "Removing common.py and unshared.py"
```

```
Deleting       examples/common.py
Deleting       examples/unshared.py

Committed revision 7.
```

Reverting Changes

Now is a good moment to examine what is happening on the file system when we delete a file. We're going to delete `__init__.py`. Don't worry too much, though—we're going to resurrect it.

```
$ ls
```

```
__init__.py     greetings
```

```
$ svn delete __init__.py
```

```
D         __init__.py
```

```
$ svn status
```

```
D      __init__.py
```

```
$ ls
```

```
greetings.py
```

The important thing to notice at this point is that the operation has already taken place on the filesystem. Subversion makes the changes to the working copy before they are committed to the repository. Your working copy is what the repository will look like after you commit your changes. Now we're going to undo those changes:

```
$ svn revert __init__.py
```

```
Reverted '__init__.py'
```

```
$ svn status
$ ls
```

```
__init__.py     greetings
```

As you can see, `__init__.py` has been restored to the working copy. This resurrected copy was pulled from the `.svn` directory contained within the working directory. The delete was also removed from the pending changes listed by `svn status`. `revert` works for all kinds of local changes, including adds, copies, moves, deletes, and modifications.

Modifying a File

Making changes to existing files is the real meat of daily work. It is not necessary to explicitly open a file in Subversion. All files are considered to be fair game for editing. We've made some changes to the file `src/examples/greetings/standard.py`. `svn status` shows that we've modified the file:

```
$ svn status
```

```
M      greetings/standard.py
```

The M indicates that the file has been modified. This is determined by comparing the working copy with the stored revision in one of the .svn directories. Because it is performed against a locally stored copy, you can run this even if you're disconnected from the server. You can find out what changes were made by using the svn diff command:

```
$ svn diff greetings/standard.py
```

```
Index: greetings/standard.py
--- greetings/standard.py       (revision 7)
+++ greetings/standard.py       (working copy)
@@ -1,6 +1,7 @@
#!/usr/bin/python
    class HelloWorld(object):
        """Simple hello world example"""

        def main(self):
           print "Hello World!"
```

The diff shows that the comment """Simple hello world example""" was added. As with the status request, the diff is done against the locally stored copy, and it can be performed even when disconnected from the server. If you were dissatisfied with the changes, you could revert them using svn revert, but you're satisfied, so you commit it:

```
$ svn commit -m "Adding doc string to HelloWorld"
```

```
Sending        examples/greetings/standard.py
Transmitting file data .
Committed revision 8.
```

Updating Your Working Copy

Outside of your local development environment there will be multiple people working with the repository. The code will be changing. The longer your project stays out of the trunk, the further it will diverge from the code in the repository. It is important to get these changes into your working copy. It is best to do this before committing changes. This is done with the svn update command.

Now suppose that someone has edited the file standard.py since you did. Another developer modified the file and it was committed as revision 9. You can find this out using the command svn status -u:

```
$ svn status -u
```

```
       *        8   src/examples/greetings/standard.py
Status against revision:      9
```

This shows that your working copy of `standard.py` is out of date. This is indicated by the * in the first column. The `8` indicates that you have revision 8, and the line `Status against revision: 9` indicates that revision 9 is the most recent revision.

You can look at the differences using `svn diff -r BASE:HEAD`. This shows all the differing files reported in `svn status -u`.

```
$ svn diff -r BASE:HEAD
```

```
Index: src/examples/greetings/standard.py
===================================================================
--- src/examples/greetings/standard.py  (working copy)
+++ src/examples/greetings/standard.py  (revision 8)
@@ -4,6 +4,8 @@
     """Simple hello world example"""

     def main(self):
+        """Someone else added a comment here"""
+
         print "Hello World!"

 if __name__ == '__main__':
```

You can pull down the most recent revision with the `svn update` command. With no arguments, this pulls down all updates to your working copy.

```
$ svn update
```

```
U    src/examples/greetings/standard.py
Updated to revision 9.
```

Conflicting Changes

Now I'll make a change to `standard.py`. It will return an exit code upon completion. The new lines are displayed in bold.

```
#!/usr/bin/python

import sys

class HelloWorld(object):
    """Simple hello world example"""
```

```
    def main(self):
        """Print message and terminate with exit code 0"""

        print "Hello World!"
        sys.exit(0)

if __name__ == '__main__':
    HelloWorld().main()
```

While this change was made, another developer submitted revision 10. Revision 10 changes the doc string for `main()`.

```
$ svn commit -m "Exit codes are explicitly returned"
```

```
Sending        src/examples/greetings/standard.py
svn: Commit failed (details follow):
svn: Out of date: '/agile2/trunk/src/examples/greetings/standard.py' in ➥
transaction '10-1'
```

This is the usual way that you'll discover something has changed. You'll try to submit and it will fail. Nothing has changed on the filesystem, though. You've just been warned that the commit couldn't happen. You can use the commands `svn status -u` and `svn diff -r BASE:HEAD` to see what has changed.

There is another command that lets you look at the changes to be committed. This command is `svn log -r BASE:HEAD`. It shows the changes between the base revision (from your last update) and the head revision in the repository:

```
phytoplankton:~/ws/agile jeff$ svn log -r BASE:HEAD
```

```
--------------------------------------------------------------------
r9 | doug | 2007-10-09 13:08:23 -0700 (Tue, 09 Oct 2007) | 1 line

Added doc string to HelloWorld.main()
--------------------------------------------------------------------
r10 | doug | 2007-10-09 13:08:25 -0700 (Tue, 09 Oct 2007) | 1 line

Updated doc string for HelloWorld.main()
--------------------------------------------------------------------
```

`svn status` will show that `standard.py` is the only file that changed, and `svn diff` will show that the comment is correct. Now you have to merge the changes together. You do this with the commands `svn update` and `svn merge`:

```
phytoplankton:/tmp/am1/src/examples/greetings jeff$ svn update
```

```
C    standard.py
Updated to revision 10.
```

This brings down the most recent changes, as before—but notice the status line for `standard.py`. It begins with `C`, which indicates a conflict. You have to resolve the changes. Subversion has created four versions of the conflicting file that will be helpful in this process.

```
phytoplankton:/tmp/am1/src/examples/greetings jeff$ ls
```

```
__init__.py          standard.py          standard.py.mine
standard.py.r10      standard.py.r9
```

- `standard.py` is the candidate merge.
- `standard.py.mine` is the version that I just made.
- `standard.py.r9` is the virgin working copy before I made my changes in `standard.py.mine`.
- `standard.py.r10` is the conflicting head revision.

The really important file here is `standard.py`, the candidate merge. The other files exist for use with external diff tools.

In the candidate merge, Subversion has spliced together your version and the head revision. Lines that have changed in one but not the other have been added to the file. The changed lines replace the unchanged lines. Generally, changes that don't overlap lines don't overlap in functionality, so simply splicing in the changed sections is a surprisingly effective algorithm for automatically merging code. The resulting code functions in most cases. In fact, it's eerie how often the merged code results in a functioning program.

The problem arises with lines that have changed in both files. There's no automatic way to merge together these conflicting blocks. When this happens, Subversion defers to the developer's judgment. The conflicting blocks of lines are both included in the merge candidate `standard.py`. They are separated with markers indicating their source. Your copy is first, and the head revision is second. It is up to you to make the appropriate changes.

```
$ more standard.py
```

```
#!/usr/bin/python

import sys

class HelloWorld(object):
    """Simple hello world example"""

    def main(self):
<<<<<<< .mine
        """Print message and terminate with exit code 0"""
=======
        """Someone updated the doc string"""
>>>>>>> .r10
```

```
        print "Hello World!"
        sys.exit(0)

if __name__ == '__main__':
    HelloWorld().main()
```

You'll edit standard.py until it looks like you want it to, and then you'll tell Subversion that the merge is complete using the command svn resolved. Once Subversion knows that you've resolved the conflicting files, you can submit the changes.

```
$ vi standard.py
```

```
[... resolve conflict manually...]
```

```
$ more
```

```
#!/usr/bin/python
...

    def main(self):
        """Print message and terminate with exit code 0"""

        print "Hello World!"
...
```

```
$ svn resolved standard.py
```

```
Resolved conflicted state of 'standard.py'
```

```
$ svn commit -m "Exit codes are explicitly returned"
```

```
Sending        greetings/standard.py
Transmitting file data .
Committed revision 11.
```

Merging code can be one of the more onerous tasks. The longer between merges, the more changes accumulate. The more changes that accumulate, the more likely conflicts are to arise. The more conflicts you have at any one time, the more work to be done when merging. The more changes that have been made, the likelier functionality is to break, too.

The key to keeping merges simple is to merge often. The agile practice of continuous integration is based on this observation. Updating your code from the code base should be done daily if not more often. Your changes to the code base should also be committed daily if not more often. This eliminates the painful and error-prone integration phase from development.

There are times when those merges, no matter how small, will result in incorrectly functioning code. A comprehensive automated test harness can catch these errors. The agile practice of comprehensive unit testing provides this safety net.

Merging using a text editor can be one of the more confusing things to be done, particularly with more than one or two conflicts. Along with reporting status information, this is an area where GUI tools and slick interfaces come into their own.

Subverting Eclipse

Eclipse talks to revision control systems. In Eclipse terminology, a project under revision control is a *shared project*. Revision control plug-ins are referred to as *team providers*. The team provider we'll be using is called Subversive. The Subversive web site is located at `www.polarion.org/index.php?page=overview&project=subversive`, and the update site is located at `www.polarion.org/projects/subversive/download/1.1/update-site/`. There are several optional components in this package that depend upon other plug-ins that you may not have installed. By default, they are selected. In order to install Subversive, you must either install these plug-ins or deselect the optional components.

■ **Note** Subversive is likely to become the standard Subversion team provider for Eclipse, and by the time you read this, it may ship with Eclipse.

Sharing Your Subverted Project

There are several ways of getting your project into Eclipse:

- Importing directly from Subversion.
- Importing a project that has already been checked out via the command line.
- Sharing a project that has already been checked out.
- Exporting your project directly to Subversion.
- Adding sharing to a project that has not been yet been checked out. (Sadly, this is broken for `file:///` URLs in Subversive.)

You're going to import your project directly from Subversion. This is the most frequent way that you'll operate. It ensures that you have a clean environment, and it's easy to do. In Chapter 2, you set up Eclipse with your test project, named agile. You imported that project into Subversion, checked it out in another location, and made a number of changes. Those changes are in Subversion, but they're not in the workspace for agile. When you import the project, you're going to choose to overwrite that project.

You can import your project directly from Subversion because the `.project` file is checked in. The `.project` file contains the name of the project. This makes it a little trickier to import multiple versions of the same project, but it goes a long way toward ensuring that every developer has a consistently named environment. There is often a deep desire to customize project names, but consistent naming becomes important in projects where many developers work together. This is particularly true with pair programming. In such situations, developers will end up working on someone else's machine at least half of the time. Having to figure out the local namings adds unnecessary hassle and often subtly frustrates one of the pair.

Importing from Subversion

Once you've installed Subversive and restarted Eclipse, you can import from the repository. Select File ➤ Import, which will bring up the Import project window, shown in Figure 3-3.

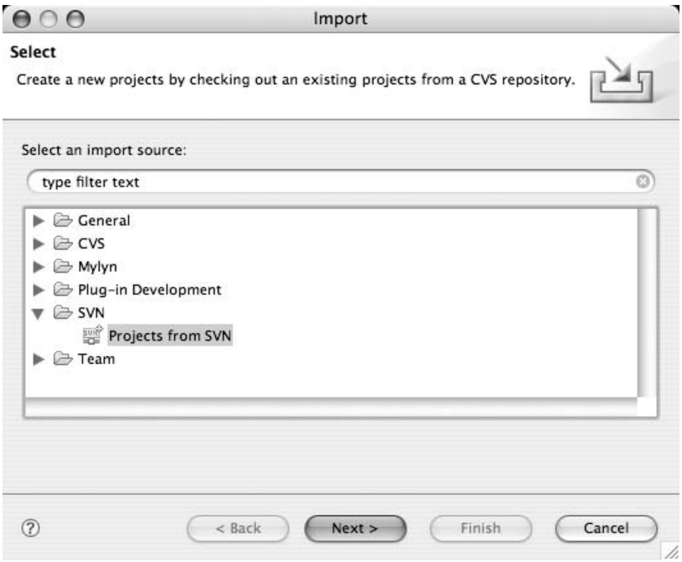


Figure 3-3. *Importing an existing project*

Select SVN ➤ Projects from SVN, and then click the Next button. This will take you to the repository selection screen, shown in Figure 3-4.

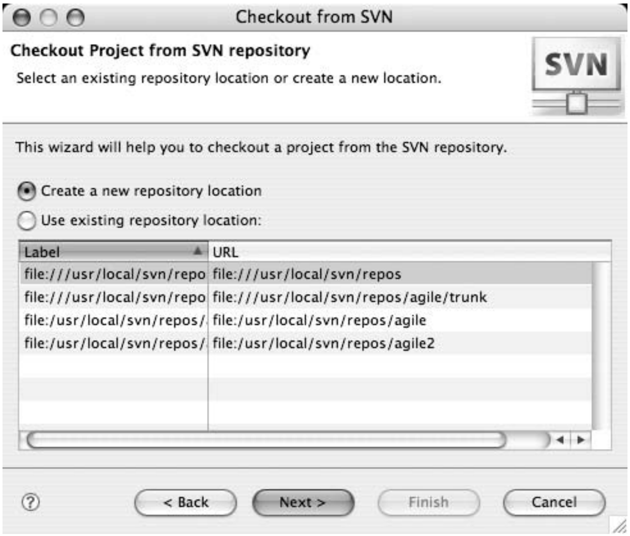


Figure 3-4. *Checkout from SVN*

If a repository had already existed, then you could select it from the list. Since you've never accessed this repository location before, you'll have to create a new one. Choose the "Create a new repository location" radio button, and click Next. This takes you to the screen shown in Figure 3-5, where you'll define the repository location.

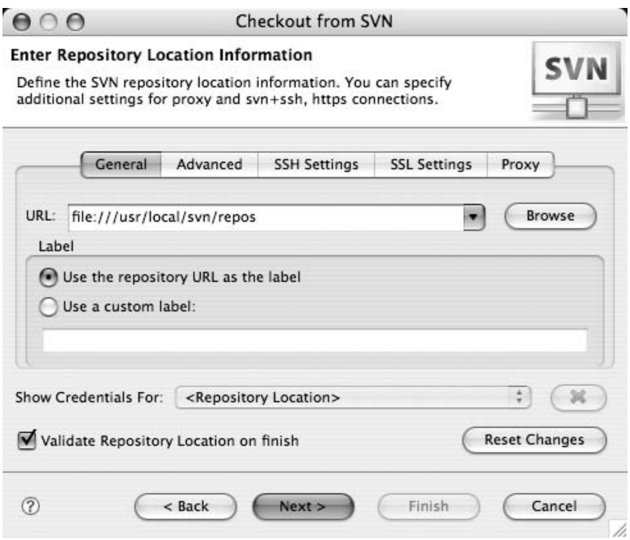


Figure 3-5. *Defining the repository location*

You should fill in the URL with the path to the local repository, which is `file:///usr/local/svn/repos`. File repositories require almost no additional information, so no extra work needs to be done. If you were using SSH, HTTP, or HTTPS as transports, then you could configure authentication information at this point. For HTTP and HTTPS, a proxy server can also be defined. The one relevant tab is labeled Advanced. It allows you to configure settings for repository structure determination.

Once you've filled in the URL as pictured, click Next, which will take you to a repository browser, as shown in Figure 3-6.

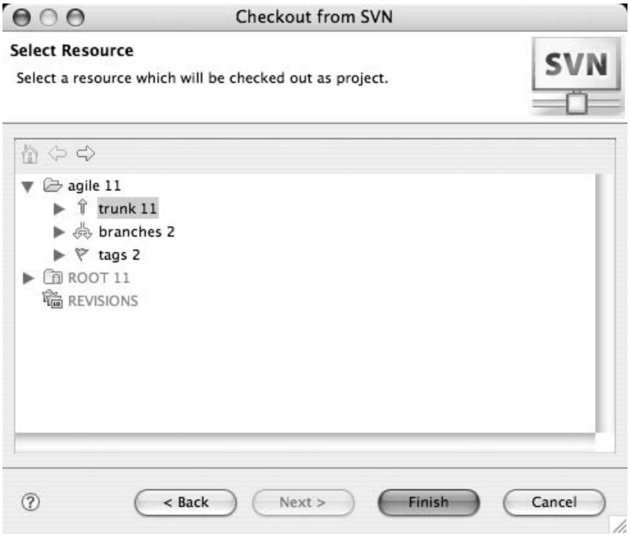


Figure 3-6. *Selecting the agile project's trunk*

Note the glyphs beside "trunk," "branches," and "tags." Subversive understands the common conventions used with Subversion. If you're importing and exporting projects, Subversion often makes the correct guess about project major vs. project minor organization. The revision numbers are beside each node.

You're going to import the trunk. Select "agile" ➤ "trunk," and then click Finish. Eclipse now gives you an opportunity to decide how you're going to import your project. The window is shown in Figure 3-7.

There are two options available. One is checking out as a project into an existing folder. This might make sense in a project where multiple repositories are being used. You might use this at a company where documentation and source code are kept in different parts of the repository or in different repositories. (I'm not a fan of separating code from documentation, but I've seen it done on more occasions than not.)

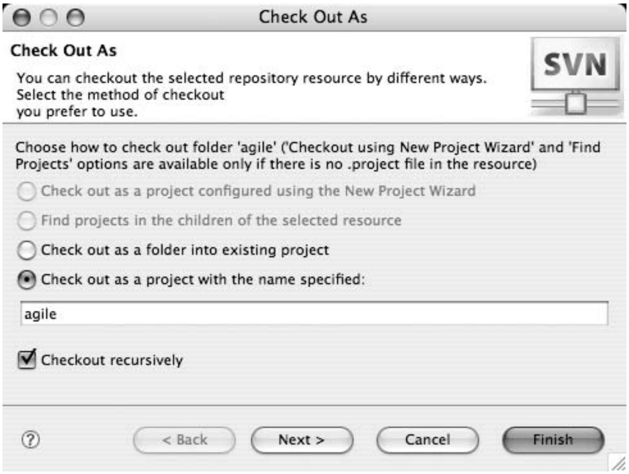


Figure 3-7. *Checking out the project*

The other option is checking out the project as a new project with the specified name. At this point, you could rename the project, but you won't be doing that. You'll choose the default name "agile" and clobber the existing project. As noted earlier, this name is extracted from the `.project` file in Subversion. Click `Finish`. Subversive detects the impending clobbering, and gives you an opportunity to back out. The open window is shown in Figure 3-8.

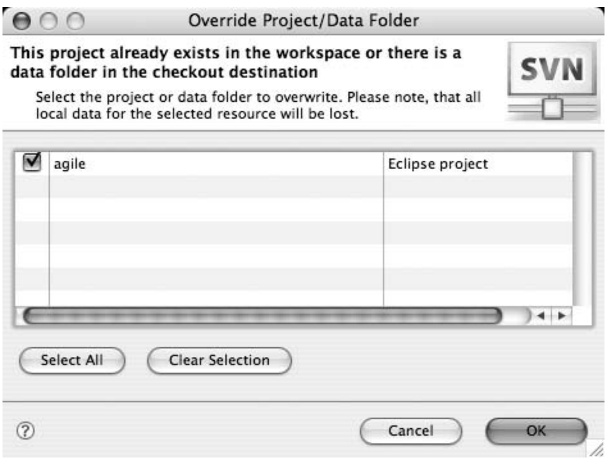


Figure 3-8. *Verifying that you want to overwrite the agile project*

At this point, you definitely want to overwrite the existing agile project. Check the "agile" check box and complete the importation by clicking OK. For a few seconds, you'll see a progress bar as the projects are reshuffled and the new project is imported. You'll then return to the workbench, as shown in Figure 3-9.

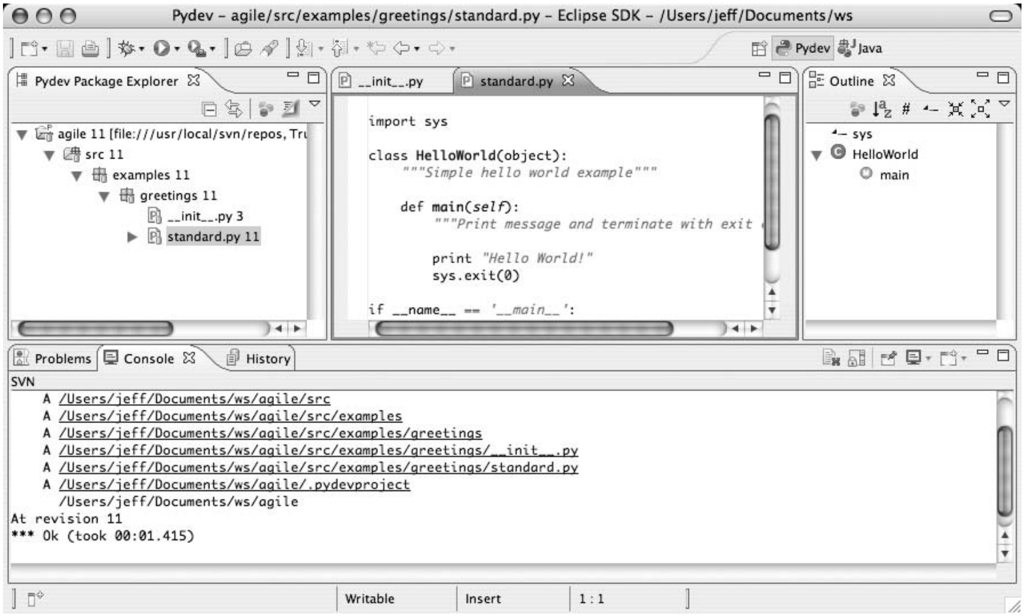


Figure 3-9. *The workbench with the agile project imported*

Activity has happened in the Console view and in the Pydev Package Explorer view. The Console view shows the output from the Subversion checkout. Subversive gives a verbose accounting of its actions. It shows both the command-line equivalent of the operation it performed and the output from the operation. You could replicate its operations on the command line if you wanted to.

The Pydev Package Explore looks different than before. Beside the project name is the URL of its repository. Beside each node is the revision number, and each icon has a small yellow glyph that indicates that the node is shared from Subversion. Other team providers use other glyphs. Additional glyphs show up to indicate other status changes. Those will be covered in the next section of this chapter.

Working with a Subverted Eclipse

Many operations in Eclipse are directly tied to Subversion. Deleting a module or package under Subversion control will delete the file from Subversion. Copying or renaming a module or package will cause the corresponding copy or move.

Surprisingly, some operations that you'd expect to be tied into Subversion are not. Creating a new file, module, or package does not automatically add the file to Subversion. More perplexingly, revert is only half done. Reverting operations that create new files will leave the newly created files in your workspace while restoring the old files. Move and rename both do this as well. These issues might be fixed by the time this you read this, however.

It's useful to be able to see what state your working copy is in with respect to the repository, and Subversive excels at this. This is done through the *team repository view*.

The Team Repository View

The team repository view supplants most of the Subversion status operations we looked at earlier in the chapter. It shows how the repository is different from your working copy, and how your working copy is different from the repository. It can combine those views showing all changes, or it can show only files with conflicts. It can show the aggregate changes, or it can break the differences down by revision.

Open the repository view by choosing the menu item Window ➤ Open View ➤ Other. This will bring up the window shown in Figure 3-10.



Figure 3-10. *Selecting a view*

Select Team ➤ Synchronize from the menu and click OK. This takes you back to the workbench, where you'll see Synchronize in the bottom pane, as in Figure 3-11. It's blocking the Console view. You can toggle back and forth between the two by clicking the tabs, but for the upcoming examples, you'll want to see both views simultaneously. Fortunately, all views in Eclipse can be moved.



Figure 3-11. *The newly opened Synchronize view*

You manipulate views by grabbing their named tabs and dragging them to a desired location. As you do this, a bounding box will show where the view will be repositioned. The other

panes in the workbench will be resized to accommodate the change. If you drag it into a list of other tabs, it will join the set. If you drag a view onto the desktop, it will become a free-floating window.

You're going to split the lower view in two. Grab the Synchronize tab and drag it all the way to the right edge. At some point, the display will show a box that splits the pane in two, and the cursor will turn into an arrow pointing to the right. Let go of the tab at that point, and you should have two panes, as shown in Figure 3-12.

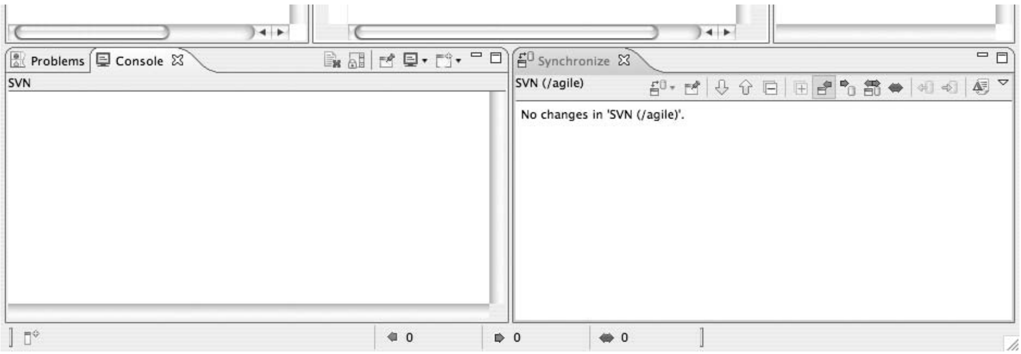


Figure 3-12. *Now the views are side by side.*

You can further adjust the proportions by grabbing the divider between the two views and dragging it left or right. You can adjust the height of both sets of views by grabbing the upper dividing bar and moving it up or down.

Notice the three colored arrows at the bottom of the screen. These only show up when the Synchronize view is active. They relate to the number of changes that have been made since the last update. The number beside the blue arrow indicates changes that have been made in the repository. The number beside the gray arrow indicates changes in your working copy. The number beside the red arrow indicates the number of places in which conflicting changes exist in both the repository and your local working copy.

The view's main area contains a tree browser showing the outstanding changes, as shown in Figure 3-13.

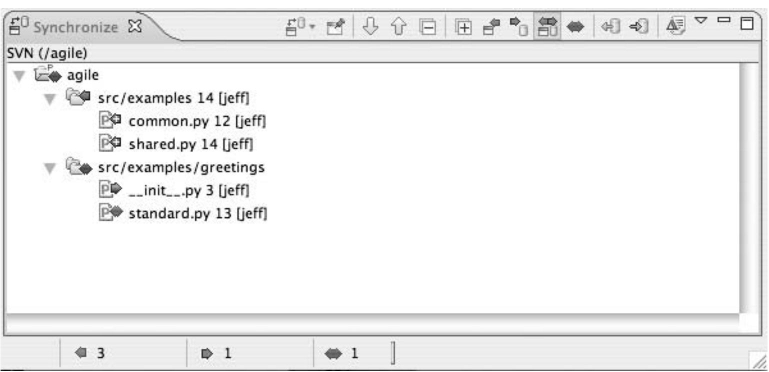


Figure 3-13. *The Synchronize view with outstanding changes*

Double-clicking a file node will bring up a diff viewer among the editor windows. The viewer shows the differences between your local copy and the repository copy. Along the top side of the bar there are a number of icons. I'll run through them from left to right:

Workbench-between-repository: This icon is entitled Synchronize SVN (/agile). When clicked, it updates the view with the most recent information from both the local working copy and the repository. The first time you do this, it also asks if you want to open the Team Synchronizing perspective. You'll be using the Synchronize view within the Pydev view, so choose No. Make sure to choose "Remember my decision" so that you won't be asked this every time you want to see what has changed. You can always open the Team Synchronizing perspective manually.

Pushpin: When active, this icon pins the window in place. Pinning is a generic Eclipse feature. Normally, a view with new information spontaneously pops to the front. Pinned views stay on top even if other views have new information.

Down arrow: This icon advances to the next displayed change. When it advances, it opens up a diff view among the editors. This view shows two versions of a file side by side. The differences between the two files are highlighted with bounding boxes. We'll look at the diff view when we get to merging later in this chapter.

Up arrow: This icon advances to the previous displayed change. Other than that, it works the same as the down arrow.

Boxed minus: This collapses all of the tree nodes in the main area of the view.

Boxed plus: This expands all of the tree nodes in the main area of the view.

Left arrow pointing to a workbench glyph: This limits the main area to new changes in the repository. When selected, the main area shows the changes that have been committed to the repository but have not been updated into the local copy. These are referred to as incoming changes.

Right arrow pointing to a repository glyph: This limits the main area to changes on the local copy. When selected, the main area shows only those changes that have been made locally but have not been committed to the repository.

Left and right arrows over workbench and repository glyphs: This icon shows all the changes that must be made. This includes both incoming changes from the repository and uncommitted local changes.

Double-ended red arrow: This indicates that you want to limit the view to files with conflicts.

Green arrow pointing away from a repository glyph: This pulls down all incoming changes.

Red arrow pointing into a repository glyph: This triggers a commit of all outstanding changes in the local copy.

Delta over a list: This alters the presentation of the view. Normally, all incoming changes are bundled together into one list. When this icon is active, the view is organized by revision number.

Upside-down white triangle: This is a standard Eclipse icon. It indicates that this view has a menu. You select the menu by clicking the icon. The menu contains a number of selections, but the two most interesting are Presentation and Schedule. Presentation allows you to select the format in which changes are presented (the default is compressed tree format). Schedule allows you to select how frequently Subversive will update the view. The default is to never update automatically. I use the Schedule option to update several times an hour for local projects and once a day for remote public projects.

Adding a File

You've already learned how to create a new Python module in Chapter 2. Create one now called `examples.common`. It doesn't matter what's in the module at the moment. Your Pydev Package Explorer should look something like Figure 3-14.



Figure 3-14. *examples.common.py created and added, but not committed*

The question mark glyph indicates that Subversion doesn't know about this file yet. You should take a look at the Synchronize view, though. You'll see that it shows up as an uncommitted change. Which of these is right? The answer is that both are right. Is there a bug? Possibly.

Subversive recognizes that the new file exists, and it assumes that you want to commit it. The Synchronize view reflects what Subversive thinks will happen. The Pydev Package Explorer reflects what Subversion reports.

If you commit from Eclipse, then the new file will be included. If you submit from the command line, then the file will not be included in the submission. My personal feeling is that Subversive should perform the add for you when it sees the new file, so that `svn commit` will check in the same files as Subversive does.

If you're just working within Eclipse, then Subversive's behavior works well. If you switch between Eclipse and the command line, then Subversive's behavior can lead to problems. Command-line tools won't know about the new files you intend to add. In that case, you should add the files explicitly.

Bring up the context menu by right-clicking `common.py` in the Package Explorer. Select Team ➤ Add to Version Control. This will bring up a window presenting a list of files to add, as shown in Figure 3-15.

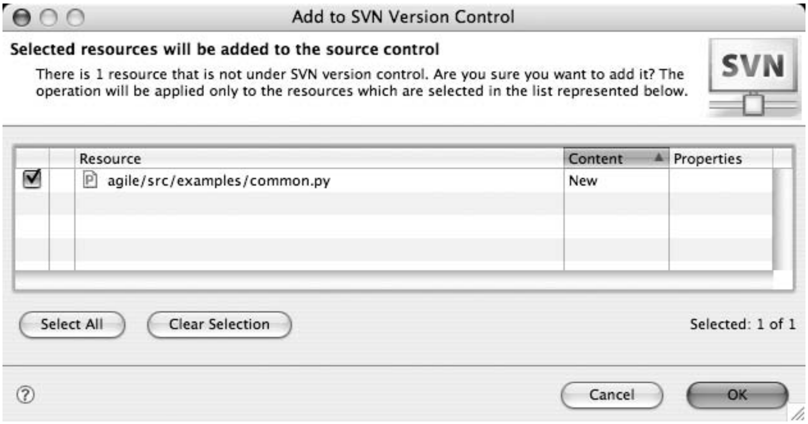


Figure 3-15. *Adding a file to Subversion*

Click the OK button. Eclipse will add all the checked files from this window; in this case, just `common.py`. The Console view should show something akin to the following messages:

```
*** Add to Version Control
svn add "/Users/jeff/ws/agile/src/examples/common.py"
A         /Users/jeff/ws/agile/src/examples/common.py
*** Ok (took 00:00.121)
```

The Pydev Package Explorer should have been altered, and should look something like Figure 3-16. There is a clock glyph on `common.py`. There is also a > preceding the name. The clock glyph indicates that the file has been scheduled for commission, and the > indicates that the node contains a change. The clock applies to just this file, but the > ripples up through the directory tree. If a directory contains a file or a directory with a change, then it is marked, too.

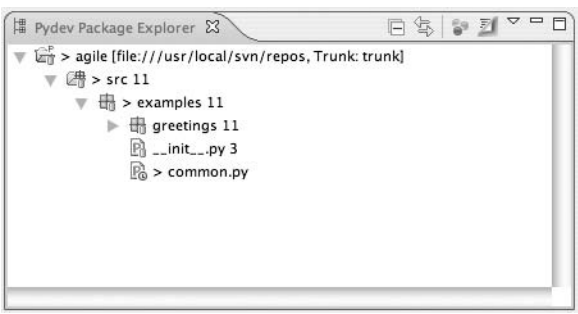


Figure 3-16. *Subversion now knows about common.py.*

Committing Changes

There are two primary ways of committing changes to Subversion. The first is by selecting individual files and using the context menu. The second is through the Synchronize view. Both paths will take you to the Commit window (shown in Figure 3-17).

Individual file selection can be done in any place that shows files. Typically, this will be through the Pydev Package Explorer or the Synchronize view. You'll use the Package Explorer for this example. Select `common.py` from the Package Explorer, right-click to bring up the context menu, and select Team ➤ Commit. This will bring up the Commit window.

The selected files will be shown in the lower portion of the window. When checked, they will be included in the commit. Instead of choosing individual files, you can also select a package or folder from the view. All new files contained within that package or folder will be selected for addition. All files contained in any subpackages or subfolders will be similarly selected.

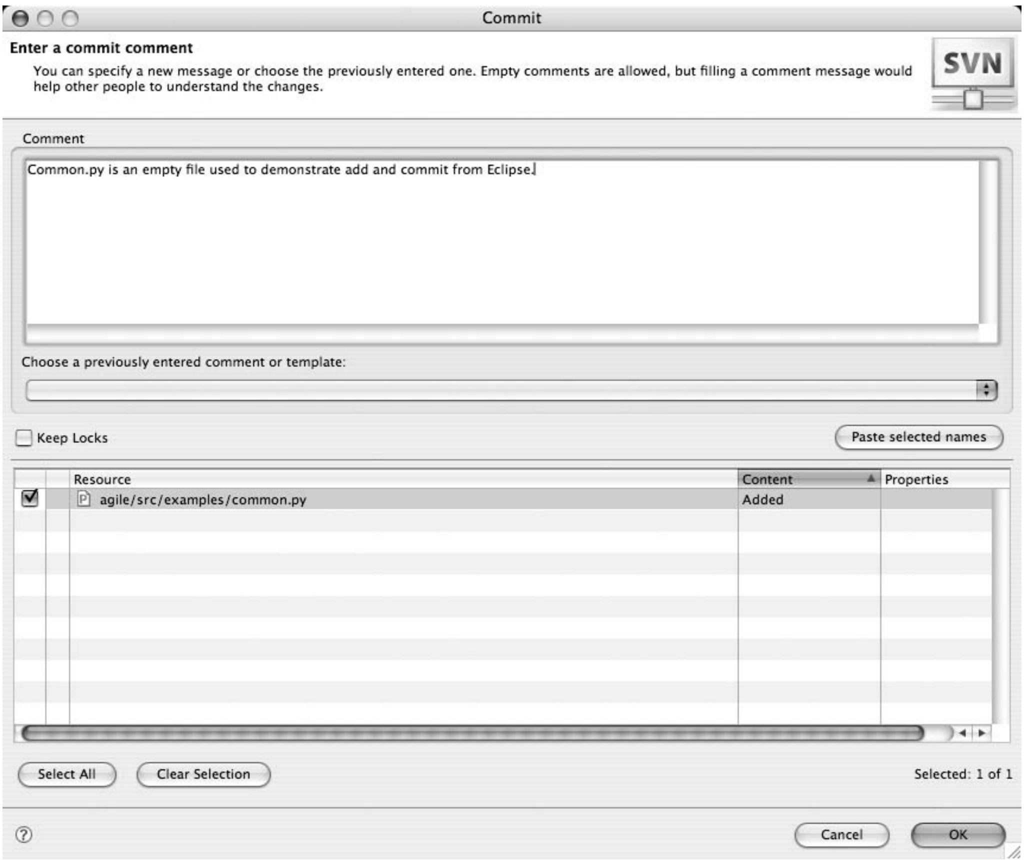


Figure 3-17. *The Commit window*

The upper portion of the window contains the commit message. You can't commit without one. There are a number of options to help with generating the message. The menu bar below allows you to choose from previously entered comments or from predefined templates. You will find the previously defined messages useful when resubmitting failed commits. Templates are useful when setting up default messages for common tasks.

Below that is the "Paste selected names" button. Clicking it will copy in the names of any files selected in the lower window. It copies those names into the comment field, one name per line. It's a real time and effort saver when you're putting together a commit message, and it helps to prevent misspellings from retyping.

When you're done composing the commit message, click OK. The console should spew out something close to the following message:

```
*** Commit
svn commit "/Users/jeff/ws/agile/src/examples/common.py" -N ➥
-m "The empty common.py file is being used to demonstrate ➥
adding and reverting."
A         ws/agile/src/examples/common.py
Transmitting file data: ws/agile/src/examples/common.py
Committed revision 12
*** Ok (took 00:01.054)
```

You can also submit all pending changes through the Synchronize view. On the left-hand side of the menu bar is a red arrow pointing at a repository glyph. Clicking this icon will bring up the Commit window (shown in Figure 3-17), but this time all scheduled changes will be included in the file list.

Editing a File

Editing is the easiest operation. Just open an editor and go to town. The complications come when it is time to submit. Subversive and the Synchronize view make it easy to anticipate conflicts, though.

You can see this by making changes to the previously added `common.py` file. Adding a doc string like `"""A sample edit"""` will suffice. Notice that when you type, there are no changes in the synchronization window. This is because you haven't saved the changes to the filesystem yet. That's indicated by a little * in the editor tab and just to the left of the file name. Once you save the file, the marker will go away, and the screen should look something like Figure 3-18.

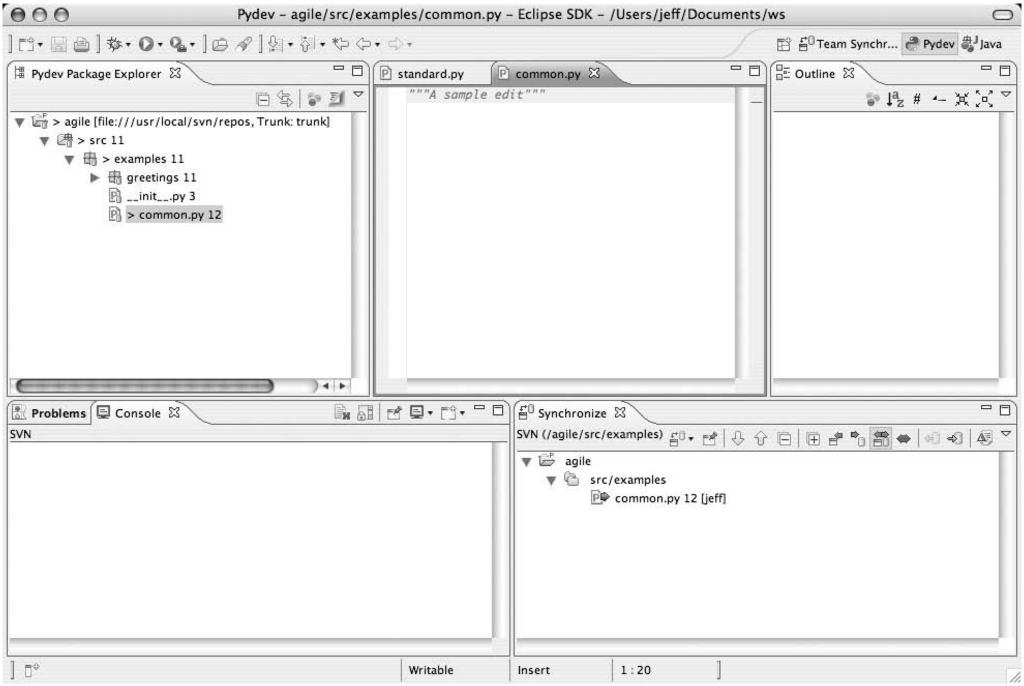


Figure 3-18. *A simple sample edit*

Within instants after you save the change, the Synchronize view will update with the modifications. Subversive also updates the Pydev Package Explorer with change markers cascading from `common.py` up through all of the containers.

Reverting Changes

You have already seen how files can be committed using the context menu. You've seen how this works from different windows and how an entire tree can be selected by choosing the parent container. The same user interface mechanisms hold true for reverting files.

You're going to revert the edits made previously. Select `common.py` from either the Pydev Package Explorer or the Synchronize view, or just right-click in the editor for `common.py`. Bring up the context menu by right-clicking, and select Team ➤ Revert. This will bring up the Revert window, shown in Figure 3-19.

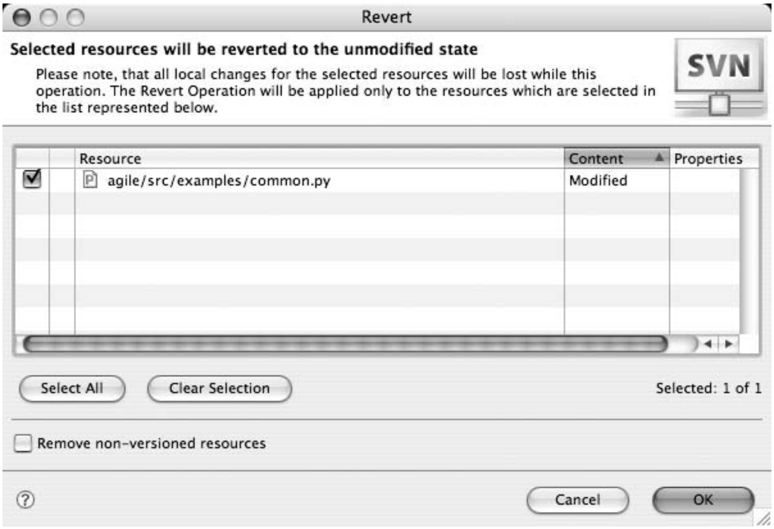


Figure 3-19. *The Revert window*

At this point, you can deselect any files that you erroneously chose. This might happen if you selected a directory containing many files. In this case, only the file you chose should be selected. Click OK, and you should see a message similar to the following:

```
*** Revert
svn revert "/Users/jeff//ws/agile/src/examples/common.py" -R ➥
Reverted /Users/jeff/ws/agile/src/examples/common.py
*** Ok (took 00:01.003)
```

Your change should be gone, and the change markers should vanish from the Pydev Package Explorer. The `common.py` editor should be blank, and the Synchronize view should report the following: `No changes in 'SVN (/agile/src/examples)'`.

Resolving Conflicts

Suppose that someone else has made a change while you were editing `standard.py`. Once again, you've both changed the comment line for `standard.HelloWorld.main()`. This time, the other user has committed their change before you have. When you attempt to submit, you see the window shown in Figure 3-20.

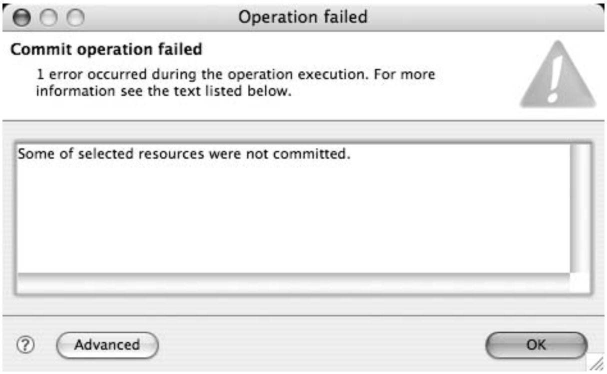


Figure 3-20. *The commit operation failed.*

Clicking the Advanced button will show the details of the failure. The message that follows shows up more or less identically in both the failure details and the Console view:

```
*** Commit
svn commit "/Users/jeff/ws/agile/src/examples/greetings/standard.py" ➥
 -m "Updating doc string."
M        /Users/jeff/ws/agile/src/examples/greetings/standard.py
Transaction is out of date
svn: Commit failed (details follow):
svn: Out of date: '/agile/trunk/src/examples/greetings/standard.py' ➥
in transaction '13-1'

*** Error (took 00:01.122)
```

Here, you can see that you're in conflict with transaction 13. You can get more information from the Synchronize view, but you need to refresh it. You can do this by clicking the leftmost icon in the view's toolbar, after which the view will update and show one conflict (see Figure 3-21).

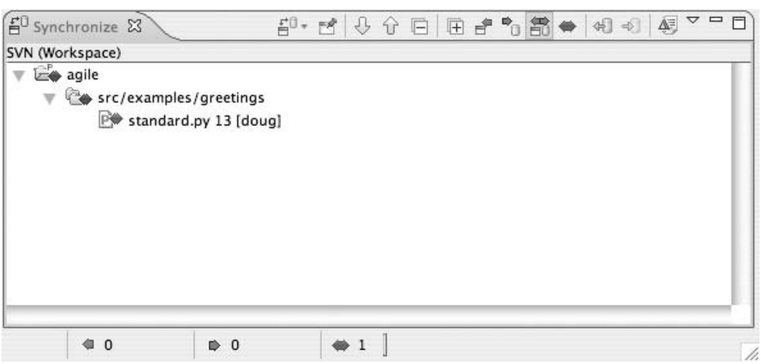


Figure 3-21. *One conflict shown in the Synchronize view*

The conflicting file is revision 13, and you can see that it was committed by doug. Double-clicking the file name will bring up the Text Compare editor, as shown in Figure 3-22. You can see from Figure 3-22 that there is a difference in whitespacing on one line.

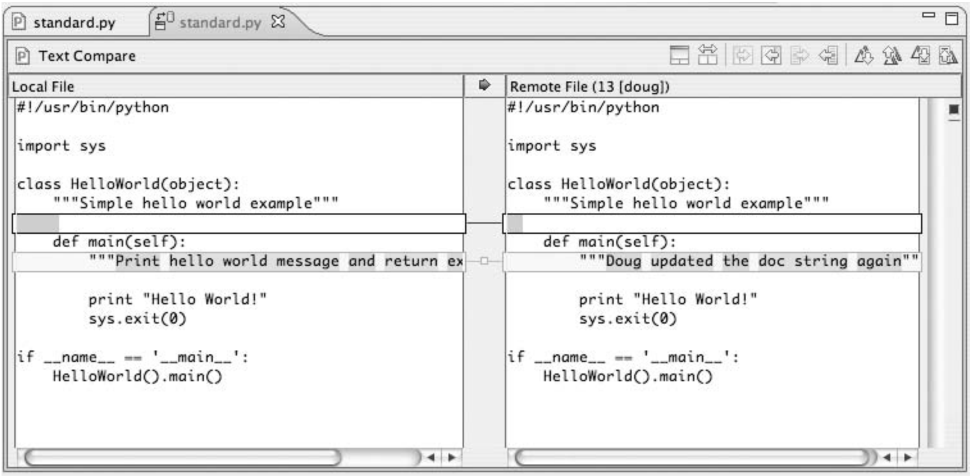


Figure 3-22. *Showing conflicts between files*

You can update your code in one of two ways. You can choose a directory tree from an explorer and then select Team ➤ Update from the context menu to bring over a subset of changes, or you can use the update button in the Synchronize view to bring over all the changes. The update button is the third icon from the right. The icon is a little green arrow pointing away from a repository glyph. After you click the button, you'll be asked to confirm that you really want to bring over these changes. You do, so you can agree. A lot happens at this point. First, the update log messages appear in the console:

```
*** Update
svn update "/Users/jeff/ws/agile/src/examples/greetings/standard.py" -r HEAD
C         /Users/jeff/ws/agile/src/examples/greetings/standard.py
At revision 13
*** Warning (took 00:00.584)
```

Subversion creates the four versions of the changed file, as in the command-line example. These show up in the Pydev Package Explorer view. In that same view, the repository glyph next to `standard.py` turns red to indicate that the file is in conflict. The Text Compare editor loads the candidate merge of `standard.py`, and you can see the conflict markers. The conflict markers prevent the file from parsing as legal Python, so you'll see a chain of red error marker glyphs on the lower left-hand corners of each node in the Synchronize view. You can see all of this in Figure 3-23 (if you look closely).

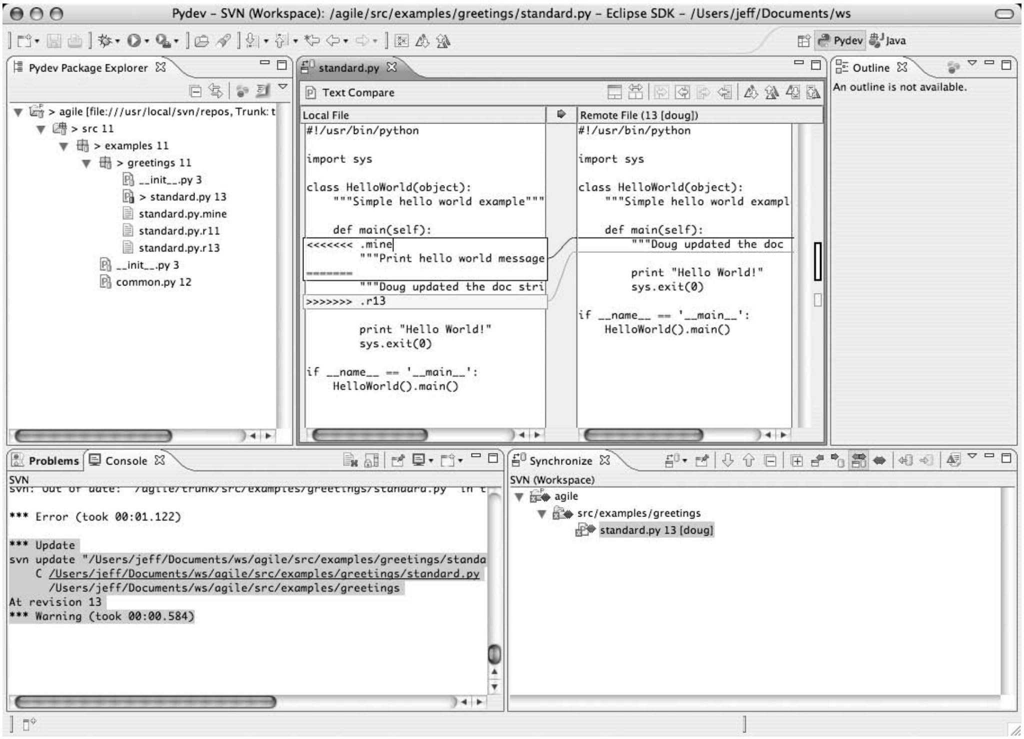


Figure 3-23. *The workbench with conflicting files*

Subversive understands that the three additional files are related to the conflict. It does not treat them as outbound changes, and they do not show up in the Synchronize view.

Open up an editor for `standard.py` and make the desired changes. After the changes are made, it's time to mark the file as resolved. This can be done from the context menu in an explorer view or from the context menu in the Synchronize view. From an explorer, the menu option is Team ➤ Mark as Merged, and from the Synchronize view, the menu option is Mark as Merged—either choice works. After one is selected, Eclipse will grind away for a second or two. You'll see a progress bar, and afterward the conflicts will disappear from the Synchronize view to be replaced by a normal pending update marker.

At this point, you can safely commit the changes. Your previous commit comment will be accessible from the drop-down menu on the Commit window, so there is no need to retype it, although you will have an opportunity to edit it.

Deleting Files

Files can be selected for deletion using pretty much any tree browser or through the file's editor window. Selecting a directory will delete all of its contents, too. File selection is exactly the same as with adding, reverting, or committing. It's only a little different when you choose to delete the selection.

Deleting files can be done in three ways. One is to bring up the context menu and select Delete. Another is to select Edit ➤ Delete from the main menu. Finally, you can press the Delete key.

Once you confirm your deletion, the files will be removed. You should see a log message in the console similar to this:

```
*** Delete
svn delete "/Users/jeff/ws/agile/src/examples/common.py" ➥
--force
D         /Users/jeff/ws/agile/src/examples/common.py
*** Ok (took 00:00.099)
```

At this point, the selection is scheduled for deletion, and it should show up in the Synchronize view. The icon beside the file name indicates the scheduling. It is a little outbound arrow glyph containing a minus sign indicating that the file will be removed.

Moving Files

Files and directories can be moved from one directory to another. This is done by selecting the candidate files from an explorer and either choosing Move from the context menu or Edit ➤ Move from the main menu. This will bring up a file browser to select the destination directory.

When Eclipse moves the files, Subversive will schedule a series of adds and deletes to perform the move. The message for moving a single file should look similar to the following:

```
*** Move
A         /Users/jeff/ws/agile/src/examples/greetings/common.py
D         /Users/jeff/ws/agile/src/examples/common.py
*** Ok (took 00:01.093)
```

The add operation maintains history between the original files and the new files. Each add and delete will show up in the Synchronize view.

Renaming Files

Only one file at a time can be renamed. You can select a file from an explorer view, and then you can choose either Rename from the context menu or Edit ➤ Rename from the main menu. This brings up a window that allows you to select a new name. Subversive treats the rename exactly as it treats a move; a similar message shows up in the console window:

```
** Move
A         /Users/jeff/ws/agile/src/examples/uncommon.py
D         /Users/jeff/ws/agile/src/examples/common.py
*** Ok (took 00:00.152)
```

The difference between copying and renaming is just the interface to the command.

Copying Files

Normally in Eclipse, you use the copy and paste operations to copy files. This is a no-no when using Subversion and many other source control systems. Subversion needs to maintain the history of a copied file. Because copy and paste are separate operations, that information is lost. Subversive gets around this by providing a special copy operation in the Team menu.

Copying files from one directory to another is much like moving files. The files to be copied are selected from an explorer, and then the copy operation is selected. From the context menu, you select Team ➤ Copy To, which brings up the screen shown in Figure 3-24.



Figure 3-24. *Copying a file*

The destination is selected from the tree browser as when moving files, but there are several other options. As with move, you select a destination directory. Subversive normally uses the Subversion `copy` command, which tracks history. You can see this in the console:

```
*** Copy
svn copy "/Users/jeff/ws/agile/src/examples/common.py" "/Users/jeff/ws/agile/➥
src/common.py"
A         /Users/jeff/ws/agile/src/common.py
*** Ok (took 00:01.330)
```

You can specify a new name at this point. If you are copying a single file and you rename it, then the destination file will be placed into the destination directory with the new name. If you are copying more than one file and you specify a new name, then something else happens. A directory with the new name is created in the destination directory. The files are then copied into this new subdirectory. You can see this in the console as `src/examples/__init__.py` and `src/examples/common.py` are copied into `src` and renamed to `stuff`:

```
*** Copy
svn add "/Users/jeff/ws/agile/src/stuff" -N ➥
A         /Users/jeff/ws/agile/src/stuff
svn copy "/Users/jeff/ws/agile/src/examples/__init__.py" ➥
"/Users/jeff/ws/agile/src/stuff/__init__.py"
A         /Users/jeff/ws/agile/src/stuff/__init__.py
svn copy "/Users/jeff/ws/agile/src/examples/common.py" ➥
"/Users/jeff/ws/agile/src/stuff/common.py"
A         /Users/jeff/ws/agile/src/stuff/common.py
*** Ok (took 00:01.878)
```

You can look at this as an obscure way of copying files into a new directory.

You can also turn off resource history. Normally, Subversive tracks the parentage between the original and its copy. Most of the time you want to keep this information, but there are times when you don't; for example, you might be using a piece of example or demo code as a starting point for real work. You don't really care that you started with the example code. In these cases, you can turn off the resource history.

If you copy a single file without revision history, then the file is added to the filesystem in the new destination. You will see no Subversion messages in the console. If you rename a single file without revision history, then it will be added to the filesystem with the new name. You will see no Subversion messages in the console. If you try to rename multiple files without revision history, then something bad happens. You'll see an attempted copy message and an error message in the console. As of Subversive 1.1.7, copying multiple files while renaming without revision history is broken. Hopefully, this case will be fixed by the time you read this.

Reverting Moves, Renames, and Copies

Reverting a move, rename, or a copy doesn't work the way you might expect it to. Reverting a moved, renamed, or copied file will only undo the Subversion operations creating that file. It won't remove the file from the local filesystem. You'll have to delete the new files manually, or else they will be added during the next commit. Reverting the new file also won't undo the operations affecting the original file or files either. For a rename or move, you'll have to revert the deletes manually. Reverting an entire directory or using the Synchronize view makes things easier.

Summary

There is no reason not to use a revision control system. There is no reason to lose code. Revision control systems are common and many are free. They provide a shared repository that allows you to look at your code at any point in time. They should serve as the shared repository for all development sources on any project. All developers submit their changes to the repository, and they receive one another's changes through the repository.

Although there are many revision control systems out there, I've focused on Subversion. Subversion is a free revision control system based around the edit-and-merge paradigm. It supports network access, a topic that will be examined in more detail in Chapter 5. Commits

are atomic and ordered through a global, monotonically increasing revision number. All work in Subversion is done in a local working copy until the changes are committed.

You saw how to work with Subversion from the command line and through Eclipse. This included setting up a repository, obtaining an initial set of files, and performing day-to-day operations such as adding, editing, deleting, copying, and moving files. In addition, you learned how to maintain consistency between the repository and the working copy on your local machine. Differences between the two can be examined, conflicts can be resolved, and undesired changes can be reverted.

Now that you know how to work with Subversion and Eclipse, it is time to start building a real project. There will be many problems to address. The project will need to be built, deployed, and packaged. The packages need to be tracked and versioned, and unit tests must be run over and over again. After all this, the code is deployed into the development environment to see how it interacts with the rest of the Python installation.

This could be done from scratch. Reams of Python code could be written. It might even make a fun project if you're into that sort of thing. Fortunately, though, it has already been done with a package called Setuptools. In the next chapter, I'll be showing you how to use it. It accomplishes all the things described here and more.

CHAPTER 4

Setuptools: Harnessing Your Code

This chapter focuses on replicable builds—a small but vital part of continuous integration. If a build can't be replicated, then test harnesses lose their efficacy. If one build differs from another build of the same code, then it is possible for tests to succeed against one build while failing against another, and testing loses its meaning. In the worst case, if a build can't be replicated, then it can become well-nigh impossible to diagnose and fix bugs in a consistent manner.

Avoiding manual configuration is the key to replicable builds. This isn't a slight against developers. People are prone to errors, while computers are not. Every manual step is an opportunity for error and inconsistency, and every error and inconsistency is an opportunity for the build to subtly fail. Again and again, this point will drive the design of the harness that ties the disparate pieces of the build together.

The harness will be built using the package Setuptools. Setuptools supersedes Python's own Distutils library, but as of Python 2.5, it is still a third-party package. Obtaining and installing Setuptools with Python 2.5 and earlier is demonstrated in this chapter.

Setuptools uses distributable packages called *eggs*. Eggs are self-contained packages. They fulfill a similar role to RPMs in the Linux world, or GEMs in Ruby installations. I'll describe eggs and demonstrate how to build and install them, along with the steps involved in installing binaries. The mystery of version numbering will be explained, too.

When complete, the demonstration project can be built on any machine with no more than a stock Python installation. All dependent packages are bundled with it, including Setuptools itself. The harness produced here is generic and can be used in any project. This chapter's work will prepare you for the subsequent chapter on automated builds.

The Project: A Simple RSS Reader

For the next few chapters, we're going to be building a single project. It's a simple RSS reader. RSS stands for Really Simple Syndication. It is a protocol for publishing frequently updated content such as news stories, magazine articles, and podcasts. It will be a simple command line tool showing which articles have been recently updated.

This chapter and the next don't demand much functionality—just enough to verify building and installation—so the program isn't going to be very exciting. In fact, it won't be much more than Hello World, but it will run, and throughout the book it will grow. This way of doing

things isn't just convenient for me. It also demonstrates the right way to go about developing a program.

Continuous integration demands that a program be built, installed, executed, and tested throughout development. This guarantees that it is deployable from the start. By moving deployment into the middle of the development process, continuous integration buffers the sudden shock that often arises when a product finally migrates to an operational environment.

Optimally, the build, installation, execution, and tests are performed after every commit. This catches errors as soon as they hit the source repository, and it isolates errors to a specific code revision. Since the changes are submitted at least daily, the amount of code to be debugged is kept to a minimum. This minimizes the cost of fixing each bug by finding it early and isolating it to small sets of changes.

This leads to a style of development in which programs evolve from the simplest implementation to a fully featured application. I'll start with the most embryonic of RSS readers, and I'll eventually come to something much more interesting and functional. This primordial RSS reader will be structured almost identically to the Hello World program in Chapter 3. The source code will reside in a directory called `src`, and `src` will reside in the top level of the Eclipse project.

Initially, we'll have two files: `src/rsreader/__init__.py` and `src/rsreader/app.py`. `__init__.py` is empty, and `app.py` reads as follows:

```
import sys

def main():
    print "OK" # give us some feedback
    return 0 # exit code

if __name__ == '__main__':
    sys.exit(main())
```

This project should be checked into your source repository as `svn:///usr/local/svn/repos/rsreader/trunk`.

Python Modules

Python bundles common code as packages. Python packages and modules map to directories and files. The presence of the file `__init__.py` within a directory denotes that the directory is a Python package. Each package contains child packages and modules, and every child package has its own `__init__.py` file.

Python supports multiple package trees. These are located through the Python path variable. Within Python, this variable is `sys.path`. It contains a `list` of directories. Each directory is the root of another tree of packages. You can specify additional packages when Python starts using the `PYTHONPATH` environment variable. On UNIX systems, `PYTHONPATH` is a colon-separated directory list. On Windows systems, the directories are separated by semicolons.

By default, the Python path includes two sets of directories: one contains the standard Python library or packages, and the other contains a directory called `site-packages`, in which nonstandard packages are installed. This begs the question of how those nonstandard packages are installed.

The Old Way

You've probably installed Python packages before. You locate a package somewhere on the Internet, and it is stored in an archived file of some sort. You expand the archive, change directories into the root of the unpacked package, and run the command `python setup.py install`. The results are something like this:

```
running install
running build
running build_py
running install_lib
creating /Users/jeff/Library/Python/2.5/site-packages/rsreader
copying build/lib/rsreader/__init__.py -> /Users/jeff/Library/➥
Python/2.5/site-packages/rsreader
copying build/lib/rsreader/app.py -> /Users/jeff/Library/➥
Python/2.5/site-packages/rsreader
byte-compiling /Users/jeff/Library/Python/2.5/site-packages/➥
rsreader/__init__.py to __init__.pyc
byte-compiling /Users/jeff/Library/Python/2.5/site-packages/➥
rsreader/app.py to app.pyc
running install_egg_info
Writing /Users/jeff/Library/Python/2.5/site-packages/➥
RSReader-0.1-py2.5.egg-info
```

`setup.py` invokes a standard package named Distutils, which provides methods to build and install packages. In the Python world, it fulfills many of the same roles that Make, Ant, and Rake do with other languages.

Note how the files are installed. They are copied directly into `site-packages`. This directory is created when Python is installed, and the packages installed here are available to all Python programs using the same interpreter.

This causes problems, though. If two packages install the same file, then the second installation will fail. If two packages have a module called `math.limits`, then their files will be intermingled.

You could create a second installation root and put that directory into the per-user `PYTHONPATH` environment variable, but you'd have to do that for all users. You have to manage the separate install directories and the `PYTHONPATH` entries. It quickly becomes error prone. It might seem like this condition is rare, but it happens frequently—whenever a different version of the same package is installed.

Distutils doesn't track the installed files either. It can't tell you which files are associated with which packages. If you want to remove a package, you'll have sort through the `site-packages` directories (or your own private installation directories), tracking down the necessary files.

Nor does Distutils manage dependencies. There is no automatic way to retrieve dependent packages. Users spend much of their time chasing down dependent packages and installing each dependency in turn. Frequently, the dependencies will have their own dependencies, and a recursive cycle of frustration sets in.

The New Way: Cooking with Eggs

Python eggs address these installation problems. In concept, they are very close to Java JAR files. All of the files in a package are packed together into a directory with a distinctive name, and they are bundled with a set of metadata. This includes data such as author, version, URL, and dependencies.

Package version, Python version, and platform information are part of an egg's name. The name is constructed in a standard way. The package PyMock version 1.5 for Python 2.5 on OS X 10.3 would be named `pymock-1.5-py2.5-macosx-10.3.egg`. Two eggs are the same only if they have the same name, so multiple eggs can be installed at the same time. Eggs can be installed as an expanded directory tree or as zipped packages. Both zipped and unzipped eggs can be intermingled in the same directories. Installing an egg is as simple as placing it into a directory in the `PYTHONPATH`. Removing one is as simple as removing the egg directory or ZIP file from the `PYTHONPATH`. You could install them yourself, but Setuptools provides a comprehensive system for managing them. In this way, it is similar to Perl's CPAN packages, Ruby's RubyGems, and Java's Maven.

The system includes retrieval from remote repositories. The standard Python repository is called the cheese shop. Setuptools makes heroic efforts to find the latest version of the requested package. It looks for closely matching names, and it iterates through every version it finds, looking for the most recent stable version. It searches the local filesystem and the Python repositories. Setuptools follows dependencies, too. It will search to the ends of the earth to find and install the dependent packages, thus eliminating one of the huge headaches of installing Distutils-based packages.

WHY THE CHEESE SHOP?

The cheese shop is a reference to a Monty Python sketch. In the sketch, a soon-to-be-frustrated customer enters a cheese shop and proceeds to ask for a staggering variety of cheeses, only to be told one by one that none of them are available. Even cheddar is missing.

Watching Setuptools and `easy_install` attempt to intuit the name of a package from an inaccurate specification without a version number quickly brings this sketch to mind. It helps to pass the time if you imagine Setuptools speaking with John Cleese's voice.

Setuptools includes commands to build, package, and install your code. It installs both libraries and executables. It also includes commands to run tests and to upload information about your code to the cheese shop.

Setuptools does have some deficiencies. It has a very narrow conception of what constitutes a build. It is not nearly as flexible as Make, Ant, or Rake. Those systems are configured using specialized Turing-complete programming languages. (Ant has even been used to make a simple video game.) Setuptools is configured with a Python dictionary. This makes it easy to use for simple cases, but leaves something to be desired when trying to achieve more ambitious goals.

Some Notes About Building Multiple Versions

One of the primary goals of continuous integration is a replicable build. When you build a given version of the software, you should produce the same end product every time the build is performed. And multiple builds will inevitably be performed. Developers will build the product on their local boxes. The continuous integration system will produce test builds on a build farm. A final production packaging system may produce a further build.

Each build version is tagged with a unique tag denoting a specific build of a software product. Each build is dependent upon specific versions of external packages. Building the same version of software on two different machines of the same architecture and OS should always produce the same result. If they do not, then it is possible to produce software that successfully builds and runs in one environment, but fails to build or run successfully in another. You might be able to produce a running version of your product in development, but the version built in the production environment might be broken, with the resulting defective software being shipped to customers. I have personally witnessed this.

Preventing this syndrome is a principal goal of continuous integration. It is avoided by means of replicable builds. These ensure that what reaches production is the same as what was produced in development, and thus that two developers working on the same code are working with the same set of bugs.

Most software products depend upon other packages. Different versions of different packages have different bugs. This is nearly obvious, but something else is slightly less obvious: the software you build has different bugs when run with different dependent packages. It is therefore necessary to tightly control the versions of dependent packages in your build environments. This is complicated if multiple packages are being built on the same machine. There are several solutions to the problem.

The *virtual Python* solution involves making a copy of the complete Python installation for each product and environment on your machine. The copy is made using symbolic links, so it doesn't consume much space. This works for some Python installations, but there are others, such as Apple's Mac OS X, that are far too good at figuring out where they should look for files. The links don't fool Python. Windows systems don't have well-supported symbolic links, so you're out of luck there, too.

The *path manipulation* solution is the granddaddy of them all, and it's been possible from the beginning. The `PYTHONPATH` environment variable is altered when you are working on your project. It points to a local directory containing the packages you've installed. It works everywhere, but it takes a bit of maintenance. You need to create a mechanism to switch the path, and more importantly, the installation path must be specified every time a package is added. It has the advantages that it can be made to work on any platform and it doesn't require access to the root Python installation.

I prefer the *location path* manipulation solution. It involves altering Python's search path to add local `site-packages` directories. This requires the creation of two files: the file `altinstall.pth` within the global `site-packages` directory, and the file `pydistutils.cfg` in your home directory. These files alter the Python package search paths.

On UNIX systems, the file `~/.pydistutils.cfg` is created in your home directory. If you're on Windows, then the situation is more complicated. The corresponding file is named `%HOME%/pydistutils.cfg`, but it is consulted only if the `HOME` environment variable is defined. This is not a standard Windows environment variable, so you'll probably have to define it yourself using the command `set HOME=%HOMEDRIVE%\%HOMEPATH%`.

This mechanism has the disadvantage that it requires a change to the shared site-packages directory. This is probably limited to root or an administrator, but it only needs to be done once. Once accomplished, anyone can add their own packages without affecting the larger site. The change eliminates an entire category of requests from users, so convincing IT to do it shouldn't be terribly difficult.

Python's site package mechanism is implemented by the standard site package. Once upon a time, accessing site-specific packages required manually importing the site package. These days, the import is handled automatically. A code fragment uses site to add a site package to add per-user site directories. The incantation to do this is as follows:

```
import os, site; ➥
 site.addsitedir(os.path.expanduser('~/lib/python2.5'))
```

You should add to the altinstall.pth file in the global site-packages directory. The site package uses .pth files to locate packages. These files normally contain one line per package added, and they are automatically executed when found in the search path. This handles locating the packages.

The second file is ~/.distutils.cfg (%HOME%\distutils.cfg on Windows). It tells Distutils and Setuptools where to install packages. It is a Windows-style configuration file. This file should contain the following:

```
[install]
install_lib = ~/lib/python2.5
install_scripts = ~/bin
```

On the Mac using OS X, the first part of this procedure has already been done for you. OS X ships with the preconfigured per-user site directory ~/Library/python/$py_version_short/site-packages, but it is necessary to tell Setuptools about it using the file ~/.pydistutils.cfg. The file should contain this stanza:

```
[install]
install_lib = ~/Library/python/$py_version_short/site-packages
install_scripts = ~/bin
```

On any UNIX variant, you should ensure that ~/bin is in your shell's search path.

Installing Setuptools

Setuptools is distributed as an egg. As of version 2.5, Python doesn't natively read eggs, so there is a "chicken-and-egg" problem. This can be circumvented with a bootstrap program named ez_setup.py, which is available at http://peak.telecommunity.com/dist/ez_setup.py. Once downloaded, it is run as follows:

```
$ python ez_setup.py
```

```
Downloading http://pypi.python.org/packages/2.5/s/setuptools/➥
setuptools-0.6c7-py2.5.egg
Processing setuptools-0.6c7-py2.5.egg
Copying setuptools-0.6c7-py2.5.egg to /Users/jeff/Library/Python/2.5/site-packages
```

```
Adding setuptools 0.6c7 to easy-install.pth file
Installing easy_install script to /Users/jeff/binInstalling easy_install-2.5➥
script to /Users/jeff/bin
Installed /Users/jeff/Library/Python/2.5/site-packages/➥
setuptools-0.6c7-py2.5.egg
Processing dependencies for setuptools==0.6c7
Finished processing dependencies for setuptools==0.6c7
```

ez_setup.py uses HTTP to locate and download the latest version of Setuptools. You can work around this if your access is blocked. ez_setup.py installs from a local egg file if one is found. You copy the appropriate egg from http://pypi.python.org/pypi/setuptools using your tools of choice, and you place it in the same directory as ez_setup.py. Then you run ez_setup.py as before.

Setuptools installs a program called ~/bin/easy_install (assuming you've created a local site-packages directory). From this point forward, all Setuptools-based packages can be installed with easy_install, including new versions of Setuptools. You'll see more of ez_setup.py later in this chapter when packaging is discussed.

Getting Started with Setuptools

Setuptools is driven by the program setup.py. This file is created by hand. There's nothing special about the file name—it is chosen by convention, but it's a very strong convention. If you've used Distutils, then you're already familiar with the process. Setuptools just adds a variety of new keywords. The minimal setup.py for this project looks like this:

```
from setuptools import setup, find_packages
setup(
    # basic package data
    name = "RSReader",
    version = "0.1",

    # package structure
    packages=find_packages('src'),
    package_dir={'':'src'},
)
```

A minimal setup.py must contain enough information to create an egg. This includes the name of the egg, the version of the egg, the packages that will be contained within the egg, and the directories containing those packages.

The name attribute should be unique and identify your project clearly. It shouldn't contain spaces. In this case, it is RSReader.

The version attribute labels the generated package. The version is not an opaque number. Setuptools goes to great lengths to interpret it, and it does a surprisingly good job, using it to distinguish between releases of the same package. When installing from remote repositories, it determines the most recent egg by using the version; and when installing dependencies, it uses the version number to locate compatible eggs. Code can even request importation of a specific package version.

In general, version numbers are broken into development and release. Both 5.6 and 0.1 are considered to be base versions. They are the earliest released build of a given version. Base versions are ordered with respect to each other, and they are ordered in the way that you'd expect. Version 5.6 is later than version 1.1.3, and version 1.1.3 is later than version 0.2.

Version 5.6a is a development version of 5.6, and it is earlier than the base version. 5.6p1 is a later release than 5.6. In general, a base version followed by a string between *a* and *e* inclusive is considered a development version. A base version followed by a string starting with *f* (for *final*) or higher is considered a release version later than the base version. The exception is a version like 5.6rc4, which is considered to be the same as 5.6c4.

There is another caveat: additional version numbers after a dash are considered to be development versions. That is, 5.6-r33 is considered to be earlier than 5.6. This scheme is typically used with version-controlled development. Setuptools's heuristics are quite good, and you have to go to great lengths to cook up a version that it doesn't interpret sensibly.

The `packages` directive lists the packages to be added. It names the packages, but it doesn't determine where they are located in the directory structure. Package paths can be specified explicitly, but the values need to be updated every time a different version is added, removed, or changed. Like all manual processes, this is error prone. The manual step is eliminated using the `find_packages` function.

`find_packages` searches through a set of directories looking for packages. It identifies them by the `__init__.py` file in their root directories. By default, it searches for these in the top level of the project, but this is inappropriate for RSReader, as the packages reside in the `src` subdirectory. `find_packages` needs to know this, hence `find_packages('src')`. You can include as many package directories as you like in a project, but I try to keep these to an absolute minimum. I reserve the top level for build harness files—adding source directories clutters up that top level without much benefit.

The `find_packages` function also accepts a list of excluded files. This list is specified with the keyword argument `exclude`. It consists of a combination of specific names and regular expressions. Right now, nothing is excluded, but this feature will be used when setting up unit tests in Chapter 8.

The `package_dir` directive maps package names to directories. The mappings are specified with a dictionary. The keys are package names, and the values are directories specified relative to the project's top-level directory. The root of all Python packages is specified with an empty string (`""`); in this project, it is in the directory `src`.

Building the Project

The simple `setup.py` is enough to build the project. Building the project creates a working directory named `build` at the top level. The completed build artifacts are placed here.

```
$ python ./setup.py build
```

```
running build
running build_py
creating build
creating build/lib
creating build/lib/rsreader
```

```
copying src/rsreader/__init__.py -> build/lib/rsreader
copying src/rsreader/app.py -> build/lib/rsreader
```

```
$ ls -lF
```

```
total 696
drwxr-xr-x   3 jeff  jeff     102 Nov  7 12:25 build/
-rw-r--r--   1 jeff  jeff    2238 Nov  7 12:14 setup.py
drwxr-xr-x   5 jeff  jeff     170 Nov  6 20:45 src/
```

Interpreting the build output is easier if you understand how Setuptools and Distutils are structured. The command `build` is implemented as a module within Setuptools. The setup function locates the command and then executes it. All commands can be run directly from `setup.py`, but many can be invoked by other Setuptools commands, and this happens here.

When Setuptools executes a command, it prints the message `running command_name`. The output shows the `build` command invoking `build_py`. `build_py` knows how to build pure Python packages. There is another build module, `build_ext`, that knows how to build Python extensions, but no extensions are built in this example, so `build_ext` isn't invoked.

The subsequent output comes from `build_py`. You can see that it creates the directories `build`, `build/lib`, and `build/lib/rsreader`. You can also see that it copies the files `__init__.py` and `app.py` to the appropriate destinations.

At this point, the project builds, but it is not available to the system at large. To install the package, you run `python setup.py install`. This installs `rsreader` into the local `site-packages` directory configured earlier in this chapter.

```
$ python setup.py install
```

```
running install
running bdist_egg
running egg_info
creating src/RSReader.egg-info
writing src/RSReader.egg-info/PKG-INFO
writing top-level names to src/RSReader.egg-info/top_level.txt
writing dependency_links to src/RSReader.egg-info/dependency_links.txt
writing manifest file 'src/RSReader.egg-info/SOURCES.txt'
writing manifest file 'src/RSReader.egg-info/SOURCES.txt'
installing library code to build/bdist.macosx-10.3-fat/egg
running install_lib
running build_py
creating build
creating build/lib
creating build/lib/rsreader
copying src/rsreader/__init__.py -> build/lib/rsreader
copying src/rsreader/app.py -> build/lib/rsreader
creating build/bdist.macosx-10.3-fat
creating build/bdist.macosx-10.3-fat/egg
creating build/bdist.macosx-10.3-fat/egg/rsreader
```

```
copying build/lib/rsreader/__init__.py -> build/bdist.macosx-10.3-fat/egg/rsreader
copying build/lib/rsreader/app.py -> build/bdist.macosx-10.3-fat/egg/rsreader
byte-compiling build/bdist.macosx-10.3-fat/egg/rsreader/__init__.py to __init__.pyc
byte-compiling build/bdist.macosx-10.3-fat/egg/rsreader/app.py to app.pyc
creating build/bdist.macosx-10.3-fat/egg/EGG-INFO
copying src/RSReader.egg-info/PKG-INFO -> build/bdist.macosx-10.3-fat/egg/➥
EGG-INFO
copying src/RSReader.egg-info/SOURCES.txt -> build/bdist.macosx-10.3-fat/egg/➥
EGG-INFO
copying src/RSReader.egg-info/dependency_links.txt -> build/bdist.macosx-10.3-fat/➥
egg/EGG-INFO
copying src/RSReader.egg-info/top_level.txt -> build/bdist.macosx-10.3-fat/egg/➥
EGG-INFO
zip_safe flag not set; analyzing archive contents...
creating dist
creating 'dist/RSReader-0.1-py2.5.egg' and adding 'build/bdist.macosx-10.3-fat/➥
egg' to it
removing 'build/bdist.macosx-10.3-fat/egg' (and everything under it)
Processing RSReader-0.1-py2.5.egg
Copying RSReader-0.1-py2.5.egg to /Users/jeff/Library/Python/2.5/site-packages
Adding RSReader 0.1 to easy-install.pth file

Installed /Users/jeff/Library/Python/2.5/site-packages/RSReader-0.1-py2.5.egg
Processing dependencies for RSReader==0.1
Finished processing dependencies for RSReader==0.1
```

You can see that install invokes four commands: bdist_egg, egg_info, install_lib, and build_py:

```
running install
running bdist_egg
running egg_info
creating src/RSReader.egg-info
...
installing library code to build/bdist.macosx-10.3-fat/egg
running install_lib
running build_py
creating build
...
```

install uses bdist_egg to produce a binary distribution for the package. bdist_egg calls egg_info and install_lib. The latter in turn calls build_py to produce a new build of the package to be bundled and installed.

egg_info produces a description of the egg. Among the files produced by egg_info are a list of dependencies and a manifest listing all the files in the egg. install_lib takes the products of build_py and copies them into an assembly area where they are finally packaged up by bdist_egg. In the very end, the egg is moved into place by install.

When the process is complete, you're left with a new `dist` directory at the top level. This contains the newly constructed egg file along with any previously constructed versions.

Each step can be invoked from the command line, and all can be configured independently. This is done through a file called `setup.cfg`. Later in this chapter, this file will be used to modify installation locations.

Installing Executables

The RSReader application has been installed into `site-packages`. It can be executed with Python using the `-m` option, as in the previous section. What you want is an executable. Executables are specified in `setup.py` with entry points, which can also specify rendezvous points for plug-ins.

The `entry_points` attribute describes the entry points. It is a dictionary of lists. The keys denote the kind of entry point, and the values name entry points and map each of them to a Python function. Executables are denoted with the `console_scripts` and `gui_scripts` keys. `setup.py` now looks like this:

```
from setuptools import setup, find_packages
setup(
    # basic package data
    name = "RSReader",
    version = "0.1",

    # package structure
    packages=find_packages('src'),
    package_dir={'':'src'},

    # install the rsreader executable
    entry_points = {
        'console_scripts': [
                'rsreader = rsreader.app:main'
                ]
    },
)
```

This `entry_points` stanza installs one executable. It will be named `rsreader` on UNIX systems. On Windows systems, it will be named `rsreader.exe`. Running this program will execute the function `rsreader.app.main()`. Note that the definition contains a colon between the package path and the function name.

The executable will be installed into the Python scripts directory `~/bin` as configured in `~/.distutils.cfg`. The location is reported in the output of `python setup.py install`:

```
$ python setup.py install
```

```
running install
running bdist_egg
...
Copying RSReader-0.1-py2.5.egg to /Users/jeff/Library/Python/2.5/site-packages
```

```
RSReader 0.1 is already the active version in easy-install.pth
Installing rsreader script to /Users/jeff/bin

Installed /Users/jeff/Library/Python/2.5/site-packages/RSReader-0.1-py2.5.egg
Processing dependencies for RSReader==0.1
Finished processing dependencies for RSReader==0.1
```

Dependencies

Setuptools manages dependencies. It locates appropriate versions of dependent packages, downloads them, and installs them. It searches and retrieves them from remote or local sources.

Dependencies are managed with the `external_requirements` attribute, which is a list of dependency expression strings. The simplest dependency expression is an unadorned package name. Setuptools then searches for the latest version of that package. The meaning of "latest" is determined using the rules described in the "Getting Started with Setuptools" section earlier in this chapter.

More complex dependency expressions have a package name on the left-hand side, a version on the right-hand side, and a comparison operator between them. The expression `docutils >= 3.4` means, "Get package Docutils version 3.4 or later." Reproducibility is the primary goal, so this project will demand specific versions.

```
from setuptools import setup, find_packages
setup(
    # basic package data
    name = "RSReader",
    version = "0.1",

    # package structure
    packages=find_packages('src'),
    package_dir={'':'src'},

    # install the rsreader executable
    entry_points = {
        'console_scripts': [
                'rsreader = rsreader.app:main'
                ]
    },
    install_requires = [
                        'docutils == 0.4',
                        ],
)

$ python setup.py install
```

```
running install
running bdist_egg
...

Installed /Users/jeff/Library/Python/2.5/site-packages/RSRead➥
er-0.1-py2.5.egg
Processing dependencies for RSReader==0.1
Searching for docutils==0.4
Reading http://pypi.python.org/simple/docutils/
Reading http://docutils.sourceforge.net/
Best match: docutils 0.4
Downloading http://prdownloads.sourceforge.net/docutils/docu➥
tils-0.4.tar.gz?download
Processing docutils-0.4.tar.gz
Running docutils-0.4/setup.py -q bdist_egg --dist-dir /tmp/easy_install-ebwmnZ/➥
docutils-0.4/egg-dist-tmp-UqTwxP
"optparse" module already present; ignoring extras/optparse.py.
"textwrap" module already present; ignoring extras/textwrap.py.
zip_safe flag not set; analyzing archive contents...
docutils.parsers.rst.directives.misc: module references __file__
docutils.writers.html4css1.__init__: module references __file__
docutils.writers.newlatex2e.__init__: module references __file__
docutils.writers.pep_html.__init__: module references __file__
docutils.writers.s5_html.__init__: module references __file__
Adding docutils 0.4 to easy-install.pth file
Installing rst2html.py script to /Users/jeff/bin
Installing rst2latex.py script to /Users/jeff/bin
Installing rst2newlatex.py script to /Users/jeff/bin
Installing rst2pseudoxml.py script to /Users/jeff/bin
Installing rst2s5.py script to /Users/jeff/bin
Installing rst2xml.py script to /Users/jeff/bin

Installed /Users/jeff/Library/Python/2.5/site-packages/docu➥
tils-0.4-py2.5.egg
Finished processing dependencies for RSReader==0.1
```

The first line displayed in bold announces that Setuptools is processing the dependencies for your RSReader package. The next shows that it is searching for the dependency you specified. It searches for the package at `pypi.python.org`. `pypi.python.org` catalogs Python modules, but it doesn't store them. It has a reference to each module's download site.

Setuptools doesn't search other catalogs, indicating that it found the package's description at `pypi.python.org`. Instead, it follows the reference to `docutils.sourceforge.net`, and there it searches for a download link to the correct file version. It finds that link, and it downloads the file from `http://prdownloads.sourceforge.net`. Take note of the URL in the output; it will be important in the next section.

Once the file is downloaded, Setuptools announces the processing of the package. The name `docutils-0.4.tar.gz` indicates that the file is a TAR archive compressed with the gzip algorithm. The output shows that it is automatically uncompressed, unpacked, and installed using Docutils's own `setup.py`. The intermediate product is stored in a temporary directory that is removed at the end of the process.

There are no required dependencies for this version of Docutils, but there are optional ones. The output indicates that these optional dependencies (`optparse` and `textwrap`) are already present. There are a few warning messages, and then a series of installation messages as a group of executables are installed. These executables convert from a text format called RST to other documentation formats.

■ Note You've seen the message `zip_safe flag not set; analyzing archive contents...` several times now. It is an advisory warning. It indicates that Setuptools is not creating a zipped egg. Although Setuptools can do this instead of producing expanded directory trees, the feature can sometimes cause problems, and by default it is turned off.

Think Globally, Install Locally

Setuptools does a great job of finding packages on the Net. Sometimes I'm frightened at how good a job it does, but sometimes it fails. An author's download server may go offline. The package might get deleted. A version you depend on may no longer be available, and instead a broken version may take its place. An author might replace one version with another subtly different version with the same version number. More frequently, your Internet connection may go south.

All of these situations have the same result: the build can't be replicated. The project may not even be buildable. These aren't academic situations either—it has happened to me within the last two weeks.

The solution uses a local copy of the dependent package. That copy is checked into source control along with the project code. Setuptools is then directed to use that copy with the `--find-links` option.

I'll create a directory in the project called `thirdparty`. The file `docutils-0.4.tar.gz` is downloaded into `thirdparty` from the URL `http://prdownloads.sourceforge.net/docutils/docutils-0.4.tar.gz?download`. This location was gleaned from `python setup.py install`'s output.

```
$ mkdir thirdparty
$ cd thirdparty
$ curl -L -o docutils-0.4.tar.gz http://prdownloads.sourceforge.net/docutils➥
/docutils-0.4.tar.gz?download
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1208k  100 1208k    0     0  72013      0  0:00:17  0:00:17 --:--:-- 73992
```

```
f$ ls -lF
```

```
total 2424
-rw-r--r--   1 jeff  jeff  1237801 Nov  8 13:34 docutils-0.4.tar.gz
```

Removing an Existing Package: Undoing Your Hard Work

To demonstrate the repository, it is necessary to remove the Docutils module over and over again. Unfortunately, this is the hardest thing to do with Setuptools—the process is only partially supported. It has three steps: the entry specifying the default version must be removed from `site-packages/easy_install.pth`, the egg must be removed from `site-packages`, and the installed binaries must be removed.

The entry in `site-packages/easy_install.pth` is removed with the command `python setup.py easy_install`:

```
$ python setup.py easy_install -m 'docutils==0.4'
```

```
running easy_install
Searching for docutils==0.4
Best match: docutils 0.4
Processing docutils-0.4-py2.5.egg
Removing docutils 0.4 from easy-install.pth file
Installing rst2html.py script to /Users/jeff/bin
Installing rst2latex.py script to /Users/jeff/bin
Installing rst2newlatex.py script to /Users/jeff/bin
Installing rst2pseudoxml.py script to /Users/jeff/bin
Installing rst2s5.py script to /Users/jeff/bin
Installing rst2xml.py script to /Users/jeff/bin

Using /Users/jeff/Library/Python/2.5/site-packages/docutils-0.4-py2.5.egg

Because this distribution was installed --multi-version, before you can➥
import modules from this package in an application, you will need to 'import➥
pkg_resources' and then use a 'require()' call similar to one of these➥
examples, in order to select the desired version:

    pkg_resources.require("docutils")  # latest installed version
    pkg_resources.require("docutils==0.4")  # this exact version
    pkg_resources.require("docutils>=0.4")  # this version or higher

Processing dependencies for docutils
Finished processing dependencies for docutils
```

The `-m` option reinstalls the package in multi-version mode. In this mode, all programs must explicitly request the version of the package they require, but it has the desired side effect of removing this package's entry from `site-packages/easy_install.pth`. It also has the

happy side effect of telling you exactly what you need to do in the next two steps. It tells you where the egg is, what scripts have been installed, and where they were installed to.

The egg is deleted:

```
$ rm -rf /Users/jeff/Library/python/2.5/site-packages/doc➥
utils-0.4-py2.5.egg
```

Then the binaries are deleted:

```
$ rm /Users/jeff/bin/rst2html.py
$ rm /Users/jeff/bin/rst2latex.py
$ rm /Users/jeff/bin/rst2newlatex.py
$ rm /Users/jeff/bin/rst2pseudoxml.py
$ rm /Users/jeff/bin/rst2s5.py
$ rm /Users/jeff/bin/rst2xml.py
```

Installing from the Local Copy

I'll demonstrate the process using the `easy_install` command. `easy_install` is the Setuptools component that installs eggs.

```
$ cd ..
$ python setup.py easy_install --find-links thirdparty 'docutils==0.4'
```

```
running easy_install
Searching for docutils==0.4
Best match: docutils 0.4
Processing docutils-0.4.tar.gz
Running docutils-0.4/setup.py -q bdist_egg --dist-dir➥
 /tmp/easy_install-LMZVog/docutils-0.4/egg-dist-tmp-Xkl2d6
"optparse" module already present; ignoring extras/optparse.py.
"textwrap" module already present; ignoring extras/textwrap.py.
zip_safe flag not set; analyzing archive contents...
docutils.parsers.rst.directives.misc: module references __file__
...

Installed /Users/jeff/Library/Python/2.5/site-packages/docutils-0.4-py2.5.egg
Processing dependencies for docutils==0.4
Finished processing dependencies for docutils==0.4
```

The output shows no external searching. Instead, the file was taken from the local repository, which is what you want. There is a problem, though: this argument only works with `easy_install`. There is no similar command-line option for the `install` command, but there is another mechanism that will work.

Fixing Options with setup.cfg

Setuptools uses an optional configuration file named `setup.cfg` to configure values for options. The values set in this file work if a command is called directly by the user or if it is called indirectly through another command.

The file `setup.cfg` lives in the root directory of the project. The format is the standard Windows option file with stanzas that have bracketed section names. The section names are the same as the commands they configure.

In this case, the command name is `easy_install`, the option name is `find-links`, and the value is `thirdparty`. This can be added to the file by hand, or via Setuptools itself:

```
$ python setup.py setopt --command easy_install --option find_links➥
 --set-value thirdparty
```

```
running setopt
Writing setup.cfg
```

```
$ cat setup.cfg
```

```
[easy_install]
find_links = thirdparty
```

The file can be edited by hand, but using Setuptools has an advantage. Setuptools merges the property setting into the existing `setup.cfg`, so existing settings made by hand or script are left intact.

The repository should be used every time `setup.py` is invoked, so the file should be put under version control and checked in. From time to time, developers may need to make temporary customizations, but these shouldn't be checked in. If such modifications are checked in, the changes are usually caught by the continuous build system (the subject of the next chapter).

Bootstrapping Setuptools

You've put a great deal of effort into setting up a new development environment easily and reproducibly. However, there is one small problem: when the project is synched down to a completely virgin Python installation, it is necessary to install Setuptools using `ez_setup.py` before you can run your build script `setup.py`.

Luckily, Setuptools provides a way around this. Copy the `ez_setup.py` program (remember `ez_setup.py` from earlier in this chapter?) into the top-level directory of the project, and add the following lines to the very top of `setup.py`:

```
from ez_setup import use_setuptools
use_setuptools(version='0.6c7')
```

At this point, running setup.py will download and install Setuptools if it isn't already present. You are still dependent on a network connection, but you can solve this problem, too. If ez_setup.py finds a Setuptools egg in the current directory, then it will install from there. Copying the Setuptools egg from site-packages into the top-level project directory will suffice:

```
$ cp -rp ~/Library/Python/2.5/site-packages/setuptools-0.6c7-py2.5.egg
```

It's possible to test the whole process by deleting everything in site-packages, and then running python setup.py install. Indeed, this is what the build system will be doing in the next chapter. After the first run, the top-level directory should look something like this:

```
$ ls -lF
```

```
drwxr-xr-x   4 jeff  jeff     136 Nov  7 13:51 build/
drwxr-xr-x   3 jeff  jeff     102 Nov  7 13:51 dist/
-rw-r--r--   1 jeff  jeff    8960 Oct 27 23:20 ez_setup.py
-rw-r--r--   1 jeff  jeff    9189 Nov  7 13:50 ez_setup.pyc
-rw-r--r--   1 jeff  jeff      40 Nov  8 23:05 setup.cfg
-rw-r--r--   1 jeff  jeff    1201 Nov  8 22:56 setup.py
-rw-r--r--   1 jeff  jeff  322831 Oct 27 23:40 setuptools-0.6c7-➥
py2.5.egg
drwxr-xr-x   5 jeff  jeff     170 Nov  7 13:51 src/
drwxr-xr-x   4 jeff  jeff     136 Nov  8 13:34 thirdparty/
```

Subverting Subversion: What Shouldn't Be Versioned

The harness has created directories and files that should not be managed by Subversion. The build and dist directories contain ephemeral build artifacts generated by Setuptools. These can be regenerated at any time. Developers expect to customize setup.cfg, and their individual changes shouldn't be checked in. Required values in the file are generated by setup.py anyhow.

These files are reported by svn status. They could be ignored, but they clutter the output, and cluttered output is hard to understand. Compiled Python files are reported, too. svn status prefixes these unversioned files with ?.

```
$ svn status
```

```
?       build
?       dist
?       ez_setup.py
?       ez_setup.pyc
?       setuptools-0.6c7-py2.5.egg
?       setup.py
?       setup.cfg
```

```
?       thirdparty/docutils-0.4.tar.gz
?       configure.py
...
```

In CVS, these would be ignored via a `.cvsignore` file. Subversion doesn't use files to track this information. Its system is based on a more general mechanism called *properties*. Properties associate bits of metadata with files and directories. The metadata consists of key/value pairs. In this case, the metadata key is `svn:ignore` and the value is a list of patterns separated by newlines.

Newlines complicate setting the property. The clearest way to set multiple values for `svn:ignore` from the command line is using a temporary file. In this example, it is `/tmp/ignore.txt`, and it has the following three lines:

```
build
dist
*.pyc
```

The `svn:ignore` property is then set with `svn propset`, and the file is fed into `svn propset` via the `-F` option. Finally, the temporary file `/tmp/ignore.txt` is deleted.

```
$ svn proplist .
$ svn propset svn:ignore -F /tmp/ignore.txt .
$ rm /tmp/ignore.txt
$ svn status
```

```
M       .
?       ez_setup.py
?       setuptools-0.6c7-py2.5.egg
?       setup.py
?       thirdparty/docutils-0.4.tar.gz
...
```

```
$ svn status --no-ignore
```

```
I       build
I       dist
I       ez_setup.pyc
```

The `svn:ignore` list is attached to the top-level project directory. Property changes must be committed just like any other file modification. Since the changes are in source control, they affect all other developers. As they update their working copies with `svn update`, the `build`, `dist`, and `*.pyc` files will disappear from Subversion's reports:

```
$ svn commit -m "added build, dist, and *.pyc to svn:ignore" .
```

```
Sending        .
Committed revision 16.
```

The Easy Way with Eclipse

The same changes can be made using Eclipse and Subversive. The file or directory to be ignored is highlighted in the Package Explorer, and the context menu is brought up. Team ➤ "Add to svn:ignore" is selected, and it brings up the window shown in Figure 4-1.

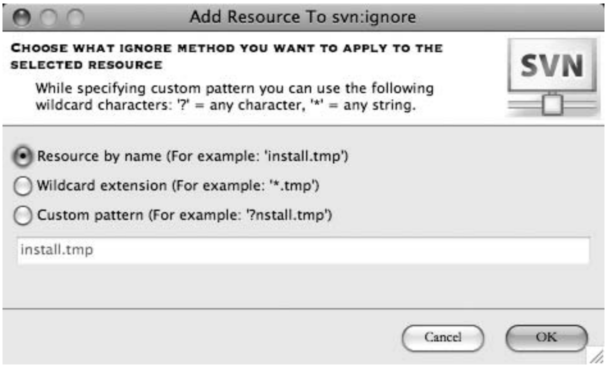


Figure 4-1. *Adding a file to svn:ignore*

The selected radio button, "Resource(s) by name," is the desired choice, as it chooses the full name of the file. Then you can click OK to add the selected file to `svn:ignore`. If you'd selected the file `ez_setup.pyc` then the "Wildcard extension" option would add the pattern `*.pyc to svn:ignore`, and the "Custom pattern" option would add the pattern specified in the text box below.

Subversive combines each new setting with the previous ones, eliminating one of the major headaches of managing `svn:ignore` from the command line. The drawback is that Subversive has no direct support for removing files from `svn:ignore`. You'll have to go back to the command line for that.

Checking in Changes: Not Losing It

At this point, quite a few changes have been made, and they need to be checked in. The files in question are `setup.py`, `ez_setup.py`, `thirdparty` (and all its contents), `configure.py`, and `setuptools-0.6c7-py2.5.egg`. These are to be placed under Subversion's control.

Working in Development Mode

RSReader was installed earlier in this chapter. Its contents are available to every other Python module in our environment, but we are actively developing it. Each change made should be available to other packages. It is possible to install the package after every change. This is feasible for small packages, but it begins to slow down development with larger packages. Besides, it's a manual step, and at some point it will be forgotten, with frustrating results.

Setuptools provides a way around this called *development mode*. Putting a package in development creates a link file in `site-packages` and a special entry in the `easy-install.pth`

file that lists the packages managed by easy_install. These redirect imports to your project directory instead of site-packages.

```
$ python -m rsreader.app
```

```
OK
```

I'll change the message printed by src/rsreader/app.py. The main entry point becomes

```
...
def main():
    print "SPAM!" # give us some feedback
    return 0 # exit code
...
```

The module is run again to demonstrate that the change has had no effect:

```
$ python -m rsreader.app
```

```
OK
```

```
$ python setup.py develop
```

```
running develop
running egg_info
writing src/RSReader.egg-info/PKG-INFO
writing top-level names to src/RSReader.egg-info/top_level.txt
writing dependency_links to src/RSReader.egg-info/dependency_links.txt
writing manifest file 'src/RSReader.egg-info/SOURCES.txt'
running build_ext
Creating /Users/jeff/Library/Python/2.5/site-packages/RSReader.egg-link (link to➥
src)
Removing RSReader 0.1 from easy-install.pth file
Adding RSReader 0.1 to easy-install.pth file

Installed /private/tmp/rsreader/src
Processing dependencies for RSReader==0.1
Finished processing dependencies for RSReader==0.1
```

The new link entry now points to the project directory. If you run the verification command, you can see that the change is picked up:

```
$ python -m rsreader.app
```

```
SPAM!
```

Finally, the print statement is changed from `SPAM!` back to `OK`, and you can verify that the change has taken effect:

```
$ python -m rsreader.app
```

```
OK
```

Summary

This chapter began with a project that will be developed throughout this book. It is a simple command-line RSS reader, implemented in only the most minimal fashion—the focus is on the harness surrounding it. This harness is constructed using Setuptools rather than the Python standard library Distutils, but the project builds from source on a machine with a stock Python installation. No work is required other than running `python setup.py install`.

Distutils and Setuptools are closely related. Distutils is the stock package for managing Python packages. It is limited in its capabilities. Setuptools is an evolution of Distutils that uses a new package format called eggs. The combination of Setuptools and eggs eliminates many shortcomings of Distutils.

Development and build environments need to be as free from interference as possible. Packages installed by other users or software on a system may cause unnoticed conflicts. This possibility is eliminated through the use of several techniques such as virtual Python installations or per-user `site-packages` and `bin` directories.

Setuptools is not a standard package, so it must be obtained and installed. Setuptools-based packages can be configured to bootstrap Setuptools, and the harness does this.

Setuptools manages external dependencies. It makes best-effort attempts at locating dependent packages. By default, it searches many sites on the Internet, but it can be configured to check local resources preferentially. Depending on third parties with potentially flaky resources is anathema to replicable builds, so the harness uses locally stored eggs. The search locations are overridden using a `setup.cfg` file. This file is checked into source control along with the rest of the project.

Sometimes it is necessary to remove a package, and doing so is fairly straightforward.

Several build and artifact directories are generated while building and installing the project. These files need to be excluded from revision control.

Setuptools's development mode links the development environment directly into `site-packages` so that changes there are directly reflected in the Python installation.

Now that I've shown you how to build and install your code using Setuptools, in the next chapter I'll show you how this can be done automatically using Buildbot.

CHAPTER 5

A Build for Every Check-In

Agile development focuses on catching bugs as soon as they are introduced. Optimally, the bugs are caught before changes are checked in, but there are classes of bugs that are expensive for the developer to verify. They happen infrequently, they are hassle to check for, and they can be painful to track down. The most visible relate to integration, platform dependencies, and external package dependencies.

The build must work on a freshly installed system, and it must contain everything that it needs to build itself. Products must frequently work with multiple versions of Python and across multiple platforms. A unit-testing module I maintain works with both Python 2.4 and 2.5, while another product I maintain is expected to work on any UNIX variant and Microsoft Windows.

Verifying these conditions before committing changes is expensive. It potentially involves many steps and a commitment of time that is guaranteed to break a programmer's flow. Just supporting one UNIX variant, Microsoft Windows, and two Python versions involves performing four sets of clean builds for every commit.

To make matters worse, most changes aren't going to cause these things to fail. With a mature product, the tests are likely to succeed dozens upon dozens of times before finally catching a failure. People aren't good at performing repetitive checks for infrequent failures—even more so when it derails their thought processes and they have to sit around waiting for the results. Eventually, vigilance lapses, a bug of this sort sneaks through, and it isn't found until deployment time. This frequently brings about a cascade of other failures.

Build servers address these problems. Rather than holding the developer responsible for verifying the correctness of the code on every system targeted, the job is given over to an automated system that is responsible for performing clean builds after every commit. Changes are validated immediately, and in case of failure, notifications are sent to the concerned parties. The build servers provide confidence that the software can always be built.

Many different build servers are available—both free and commercial. Among the more well known are CruiseControl and Anthill. This book focuses on Buildbot, an open source system written in Python. It supports build farms, in which builds are distributed to a number of client machines that then perform the builds and communicate the results back to the server. It has a centralized reporting system, and it is easily configured and extended using Python.

Buildbot Architecture

Buildbot is a common open build system. It is written in Python, but it will build anything. It uses a master-and-slave architecture. The central build master controls one or more build slaves. Builds are triggered by the master, and performed on the slaves. The slaves can be of a different architecture than the master. The slaves report build results to the master, and the master reports them to the users. The master contains a minimal web server showing the real-time build telemetry.

There are multiple options for triggering builds. The master can do it periodically, producing a nightly or hourly build. More interestingly, the master can be triggered to perform builds whenever new changes are committed.

The system demonstrated in this chapter contains a master, a slave, and a remote Subversion repository. The three systems are named `buildmaster`, `slave-lnx01`, and `source`. On my systems, these are DNS aliases for the underlying hosts. The slave performs builds against both Python 2.4 and 2.5., and these builds are triggered automatically after each commit.

Dedicated users will be created to run both Buildbot and Subversion. On the build systems, the application Buildbot will be run as the user `build`; and on the source server, Subversion will be run as the user `svn`.

ALIASING HOSTS

On my network, the names `buildmaster`, `slave-lnx01`, and `source` are aliases for the two hosts `phytoplankton` and `agile`. `buildmaster` and `source` are aliases for `phytoplankton`, and `slave-lnx01` is an alias for `agile`. The names refer to the service being provided, not the underlying host. This way, the service can be moved to another host without disrupting clients (both human and machine).

I might do this if I wanted to move `agile` to a real box rather than running it under a VM, as I currently do. I might also do this if `phytoplankton` died, or if the load of running the repository became too much for this one system to bear.

Installing Buildbot

Buildbot itself is a Setuptools package. It can be downloaded and installed using `easy_install`, but it is built on top of Twisted, which is "an event-driven networking engine." Twisted provides the bulk of the networking infrastructure for Buildbot. It's best to install Twisted before installing Buildbot.

Twisted is built with Distutils, and it must be installed carefully in multiple steps. It has its own dependency on a package called Zope Interface, which provides a limited typing system for Python. You could spend time chasing this package down, but that's not necessary, as it's bundled with Twisted. However, although it is bundled, it must be installed manually before Twisted.

I'll start by demonstrating how to install Buildbot on `buildmaster`. You'll be installing special Python installations just for the build slave's use, so it doesn't matter much where Buildbot and its dependencies are installed. I'm going to use the primary system installation:

```
$ curl -L -o Twisted-2.5.0.tar.bz2 ➥
 http://tmrc.mit.edu/mirror/twisted/Twisted/2.5/Twisted-2.5.0.tar.bz2
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 2683k  100 2683k    0     0   741k      0  0:00:03  0:00:03 --:--:--  787k
```

```
$ bunzip2 Twisted-2.5.0.tar.bz2
$ tar xvf Twisted-2.5.0.tar
```

```
Twisted-2.5.0/
Twisted-2.5.0/TwistedConch-0.8.0/
...
Twisted-2.5.0/LICENSE
Twisted-2.5.0/setup.py
```

```
$ cd Twisted-2.5.0
$ ls -F
```

```
LICENSE         TwistedMail-0.4.0/       TwistedWords-0.5.0/
README                  TwistedNames-0.4.0/     setup.py*
TwistedConch-0.8.0/     TwistedNews-0.3.0/       zope.interface-3.3.0/
TwistedCore-2.5.0/      TwistedRunner-0.2.0/
TwistedLore-0.3.0/      TwistedWeb-0.7.0/
```

```
$ cd zope.interface-3.3.0
$ sudo python ./setup.py install
```

```
running install
running bdist_egg
...
Processing dependencies for zope.interface==3.3.0
Finished processing dependencies for zope.interface==3.3.0
```

■**Warning** With most packages, running `install` correctly invokes `build`, but that has not been my experience with Twisted. It is necessary to run build and install separately.

```
$ cd ..
$ python ./setup.py install
```

```
running install
running build
...
byte-compiling /usr/lib/python2.5/site-packages/twisted/➥
news/test/test_nntp.py to test_nntp.pyc
byte-compiling /usr/lib/python2.5/site-packages/twisted/➥
plugins/twisted_news.py to twisted_news.pyc
```

At this point, Twisted is installed, and Buildbot can now be installed. Buildbot is installed with `easy_install`, which is part of Setuptools. If it hasn't been installed yet, then you'll need to do this first. See Chapter 4 for more information.

There is one catch, though. Buildbot has contributed programs that are shipped with it, but that are not installed by `easy_install`. You'll use one later, so you'll want the source package to remain on the system. The build directory option `-b` specifies a directory where the installation is staged from. When `easy_install` completes, this directory will be left behind and the component files will be accessible.

```
$ easy_install -b /tmp/bbinst buildbot
```

```
Searching for buildbot
Reading http://pypi.python.org/simple/buildbot/
...
Installed /Users/jeff/Library/Python/2.5/site-packages/➥
buildbot-0.7.6-py2.5.egg
Processing dependencies for buildbot
Finished processing dependencies for buildbot
```

```
$ buildbot --version
```

```
Buildbot version: 0.7.6
Twisted version: 2.5.0
```

The identical process must now be performed on all machines communicating with Buildbot. This includes the Subversion host, too. Once the installations are complete, the master and slave can then be configured.

Configuring the Build System

As outlined earlier, there are two build hosts in our system: the Buildbot master, named `buildmaster`, and the Buildbot slave, named `slave-lnx01`. Buildbot runs on both systems as a dedicated user, which you'll name `build`. This provides administrative and security benefits. Startup configuration can be kept within the user's account. The user `build` has limited rights, so any compromises of the Buildbot server will be limited to `build`'s account, and any misconfigurations will be limited by filesystem permissions.

When a Buildbot slave starts, it contacts the build master. It needs three pieces of information to do this. First, it needs the name of the build master so that it can find it on the network. The port identifies the Buildbot instance running on the build master, and the password authenticates the slave to the master. The master must know which port to listen on, and it must know the password that slaves will present.

In the environment discussed here, the build master runs on the host `buildmaster` listening on port `4484` for the password `Fo74gh18` from instance `rsreader-full-py2.5`. The build server instances run from within the directory `/usr/local/buildbot`. RSReader is the project started in Chapter 4. The master lives in `/usr/local/buildbot/master/rsreader`, and the slave lives in `/usr/local/buildbot/slave/rsreader`. This directory structure allows you to intermix independent Buildbot instances for different projects on the same machines.

Setting up communications between the master and the slave is the first goal. Retrieving source code or performing a build is pointless until the two servers can speak to each other.

Mastering Buildbot

The build master is configured before the slave, as the slave's status is determined through its interactions with the build master. Creating the `build` user and the directories are the first steps.

■ **Tip** If you're trying to install Buildbot on Windows systems, it is started with `buildbot.bat`. This script is installed into `\Python25\scripts`. Unfortunately, it has a hard-coded reference to a nonexistent script in `\Python23`. This reference will need to be changed by hand.

```
$ useradd buid
$ sudo mkdir -p /usr/local/buildbot/master/rsreader
$ sudo chown build:build /usr/local/buildbot/master/rsreader
```

The next steps are performed as the newly created user `build`. They create the basic configuration files for a master.

```
$ su - build
$ buildbot create-master /usr/local/buildbot/master/rsreader
```

```
updating existing installation
chdir /usr/local/buildbot/master/rsreader
creating master.cfg.sample
populating public_html/
creating Makefile.sample
buildmaster configured in /usr/local/buildbot/master/rsreader
```

```
$ ls -F /usr/local/buildbot/master/rsreader
```

```
Makefile.sample        buildbot.tac
master.cfg.sample      public_html/
```

Once upon a time (Buildbot 0.6.5 and earlier), makefiles were used to start and stop Buildbot. This mechanism has been superseded by the `buildbot` command in current versions. Makefiles can still be used to override the startup process, but that's voodoo that I won't address, so you can safely forget that `Makefile.sample` exists.

`Buildbot.tac` is only of marginally more interest. It is used by the `buildbot` command to start the server. Essentially, it defines if this server is a client or a slave. It is necessary to Buildbot's operation, but you should never have to touch it.

The `public_html` directory is the document root for the build master's internal web server. It supplies the static content that will be served to your browser. Customizations to Buildbot's appearance go here, but they are strictly optional.

Of far more interest is `master.cfg.sample`. It is the template for the file `master.cfg`, which defines most of the master's behavior. It is a Python source file defining a single dictionary named `BuildmasterConfig`. This dictionary describes almost everything about the build master and the build process that ever needs changing. Much of this chapter is devoted to writing this file.

You'll start off with a minimal `master.cfg`. It defines the `BuildmasterConfig` dictionary and aliases it to the variable `c`. This is done to improve readability and save keystrokes (although rumors of an impending keystroke shortage have been determined to be false by reputable authorities).

```
# This is the dictionary that the buildmaster pays attention
# to. We also use a shorter alias to save typing.
c = BuildmasterConfig = {}
```

Next is the `slaves` property, which defines a list of `BuildSlave` objects. Each of these contains the name of a slave and the password that will be used to secure that connection. All slaves talk to the master on a single port, and the name is necessary to distinguish them from one another. Every slave has its own password, too. A separate password allows slaves to be controlled by different individuals without compromising the security of other slaves. In our case, we have one slave named `rsreader-linux`, and its password is `Fo74gh18`.

```
####### BUILDSLAVES

from buildbot.buildslave import BuildSlave
c['slaves'] = [BuildSlave("slave-lnx01", "Fo74gh18")]
```

The master listens for connections over a single port. The `slavePortnum` property defines this. This number is arbitrary, but it should be above 1024, as lower port numbers are reserved as rendezvous locations for well-known services (like mail) and web traffic. In our configuration, it will be 4484.

```
# 'slavePortnum' defines the TCP port to listen on. This must match the value
# configured into the buildslaves (with their --master option)

c['slavePortnum'] = 4484
```

When the source code changes, a build will be triggered. The build master needs to know how to find changes. Various classes within `buildbot.changes` supply these behaviors. The class `PBChangeSource` implements a listener that sits on `slavePortnum` and waits for externally generated change notifications. When it receives an appropriate notification, it triggers a build. In a few sections, you'll configure Subversion to send these notifications.

```
####### CHANGE SOURCES

from buildbot.changes.pb import PBChangeSource
c['change_source'] = PBChangeSource()
```

The `schedulers` property defines when builds are launched. It is a list of scheduler objects. These tie a scheduling policy to a builder that actually performs the build. You're going to schedule one build for Python 2.5. The scheduler will work on any branch, and it will run when there have been no more changes for 60 seconds.

```
####### SCHEDULERS

c['schedulers'] = []
c['schedulers'].append(Scheduler(name="rsreader",
                    branch=None,
                    treeStableTimer=60,
                    builderNames=["rsreader-full-py2.5"]))
```

Build factories describe the nitty-gritty details of building the application. They construct the instructions run by slaves. I shall be spending a lot of time on build factories, but right now a simple factory will suffice to test communication between the master and slave. The simple builder factory `f1` prints the message `build was run`.

```
####### BUILDERS

from buildbot.process import factory
from buildbot.steps.shell import ShellCommand

f1 = factory.BuildFactory()
f1.addStep(ShellCommand(command="echo 'build was run'"))
```

The `builders` property contains a list of builders. A *builder* is a dictionary associating the builder's name, the slave it runs on, and a builder factory. It also names the build directory. In this case, the builder is named `buildbot-full-py2.5`, and it runs on the slave `slave-lnx01` in the directory `full-py2.5` using the builder factory `f1`. The build directory is relative to the Buildbot root. In this case, the full path to the builder will be `/usr/local/buildbot/slave/rsreader/full-py2.5`.

```
b1 = {'name': "rsreader-full-py2.5",
      'slavename': "slave-lnx01",
      'builddir': "full-py2.5",
      'factory': f,
      }

c['builders'] = [b1]
```

The `status` property controls how build results are reported. We are implementing two. The `html.WebStatus` class implements a page referred to as the *waterfall display*, which shows the entire build system's recent activity. The web server port is configured with the `http_port` keyword. Here it's being configured to listen on port 8010.

The class `mail.MailNotifier` sends e-mail when a build fails. It is inventive and persistent in its actions. There are other notification classes, with the `words.IRC` class being perhaps the most interesting of those not being used in this example.

```
####### STATUS TARGETS

c['status'] = []

from buildbot.status.html import WebStatus
c['status'].append(WebStatus(http_port=8010))

from buildbot.status.mail import MailNotifier
c['status'].append(MailNotifier(
                    fromaddr="buildbot@phytoplankton.theblobshop.com",
                    extraRecipients=["builds@theblobshop.com"],
                    sendToInterestedUsers=False))
```

The properties `projectName`, `projectUrl`, and `buildbotUrl` configure communications with the user. The project name is used on the waterfall page. The project URL is the link from the waterfall page to the project's web site. `BuildbotURL` is the base URL to reach the Buildbot web server configured in the `status` property. Buildbot can't determine this URL on its own, so it must be configured here.

```
####### PROJECT IDENTITY

c['projectName'] = "RSReader"
c['projectURL'] = "http://www.theblobshop.com/rsreader"
c['buildbotURL'] = "http://buildmaster.theblobshop.com:8010/"
```

At this point, you can start the build master:

```
$ buildbot start /usr/local/buildbot/master/rsreader
```

```
Following twistd.log until startup finished..
2008-05-12 11:21:47-0700 [-] Log opened.
…
2008-05-12 11:21:47-0700 [-] BuildMaster listening on port tcp:4484
2008-05-12 11:21:47-0700 [-] configuration update started
2008-05-12 11:21:47-0700 [-] configuration update complete
The buildmaster appears to have (re)started correctly.
```

The messages indicate that Buildbot started correctly. In previous versions, the startup messages were untrustworthy and you often had to search through the file `twistd.log` in the application directory to determine if the reported status was accurate. This seems to have been remedied as of Buildbot 0.7.7. The landing screen is shown in Figure 5-1.

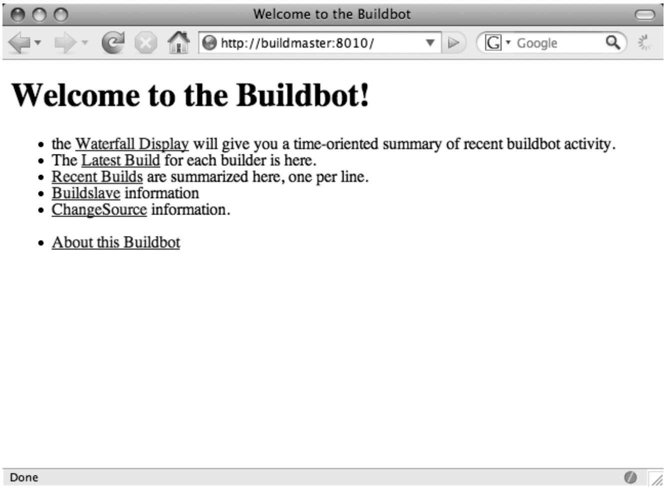


Figure 5-1. *The Buildbot landing page on the host buildmaster and port 8010*

Clicking the first link title, Waterfall Display, takes you to the page shown in Figure 5-2, which is a timeline. The top of the page represents now, and the screen extends down into the past. Each column represents a builder and the activity taking place. The builder's creation and the master's startup are both represented, so the display conveys information about the system's gross state, reducing the need to search through `twistd.log`. The red box at the top indicates that the build slave for `build-full-py2.5` is offline.

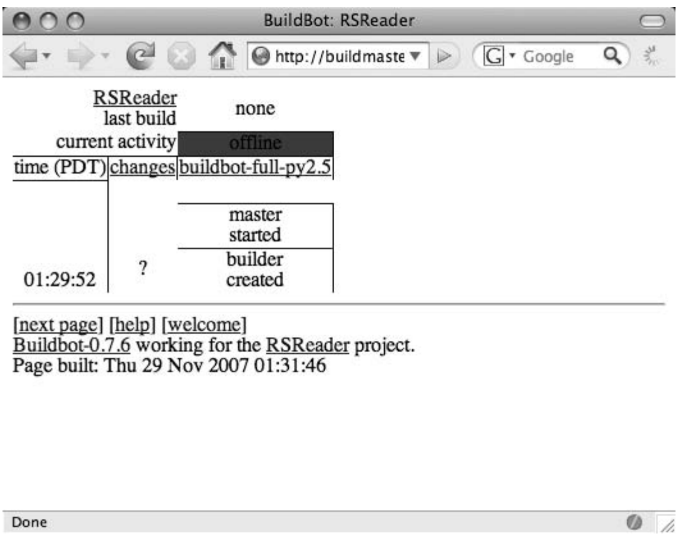


Figure 5-2. *The Buildbot waterfall display*

The properties `projectName` and `projectUrl` are used to produce the RSReader links at the top and bottom of the waterfall display. Clicking on either one is sufficient to verify the correct values. At this point, the basic server configuration is complete, and there is one last step.

The server must be started at reboot. Once upon a time, it was necessary to write a startup script and insert it into `/etc/init.d` on UNIX systems and create a few magically named symbolic links in the `/etc/rc` directories. These days, processes can be started from cron at reboot by adding the following line to the `build` user's crontab with the command `crontab -e`:

```
@reboot /path/to/buildbot start /usr/local/buildbot/master/rsreader
```

This technique should work on most modern UNIX systems, as well as Mac OS X. Cron doesn't have access to your full shell environment, so it is important to use the full path to the `buildbot` executable. Your shell may be able to locate `buildbot` when you are logged in, but it may not be able to when run from cron's extremely limited environment.

Enslaving Buildbot

In grand strokes, creating a basic Buildbot slave is similar to creating a master, but much simpler in the initial details. If running on a separate system, as in this example, then Buildbot must be installed first. Then the `build` user and Buildbot directories are created, a slave instance is created, the configuration files are updated, and Buildbot is started. In this test environment, the slave runs on `slave-lnx01`.

```
$ useradd build
$ sudo mkdir /usr/local/buildbot/slave/rsreader
$ sudo chown build:build /usr/local/buildbot/slave/rsreader
```

After creating the build directories, the client is configured from `build`'s account on `slave-lnx01`. Four pieces of information are necessary. The slave contacts the Buildbot master using the master's host name and port. In this case, the host name is `buildmaster` and the port is 4484. The slave identifies itself with a unique name. (The host name is insufficient, as there can be more than one slave running on a single host.) This is the name referred to on the master in both the builder definition and the `slaves` property. Finally, the slave needs the password to secure the connection. The `BuildSlave` object in `master.cfg` defines it; in this case, it's `Fo74gh18`.

```
$ su - build
$ buildbot create-slave /usr/local/buildbot/slave/rsreader ➥
 buildmaster:4484 rsreader-linux Fo74gh18
```

```
updating existing installation
chdir /usr/local/buildbot/slave/rsreader
creating Makefile.sample
mkdir /usr/local/buildbot/slave/rsreader/info
Creating info/admin, you need to edit it appropriately
Creating info/host, you need to edit it appropriately
Please edit the files in /usr/local/buildbot/slave/rsreader/info appropriately.
buildslave configured in /usr/local/buildbot/slave/rsreader
```

```
$ cd /usr/local/buildbot/slave/rsreader
$ ls -F
```

```
buildbot.tac info/ Makefile.sample
```

```
$ ls -F info
```

```
admin host
```

Buildbot.tac and Makefile.sample are analogous to those files on the build master. Buildbot uses Buildbot.tac to start the slave, but the slave's configuration is also in this file. Changes to the four configuration parameters can be made here. As with the master, Makefile.sample is a vestigial file lingering from previous generations of Buildbot.

The files in the info directory are of more interest. They are both text files containing information that is sent to the build master. info/admin contains this Buildbot administrator's name and e-mail address, while info/host contains a description of the slave.

The default for info/admin is Your Name Here <admin@youraddress.invalid>. In my environment, it is set to Jeff Younker <buildmaster@theblobshop.com>. The description in slave-lnx01's info/host file reads Produces pure Python 2.5 builds. info/host is just a text file, and the information is to make your life, and the life of everyone who uses your build system, a little bit brighter and clearer, so make the description concise and informative. With these changes in place, the client can be started.

```
$ buildbot start /usr/local/buildbot/slave/rsreader
```

```
Following twistd.log until startup finished..
2007/11/27 02:18 -0700 [-] Log opened.
2007/11/27 02:18 -0700 [-] twistd 2.5.0 (/usr/bin/python 2.5.0) starting up
2007/11/27 02:18 -0700 [-] reactor class: ➥
<class 'twisted.internet.selectreactor.SelectReactor'>
2007/11/27 02:18 -0700 [-] Loading buildbot.tac...
2007/11/27 02:18 -0700 [-] Creating BuildSlave
2007/11/27 02:18 -0700 [-] Loaded.
2007/11/27 02:18 -0700 [-] Starting factory <buildbot.slave.bot.BotFactory➥
instance at 0xa0e484c>
2007/11/27 02:18 -0700 [broker,client] message from master: attached
The buildslave appears to have (re)started correctly.
```

The build slave has started and connected to the build master, which you can see on the waterfall display in Figure 5-3.

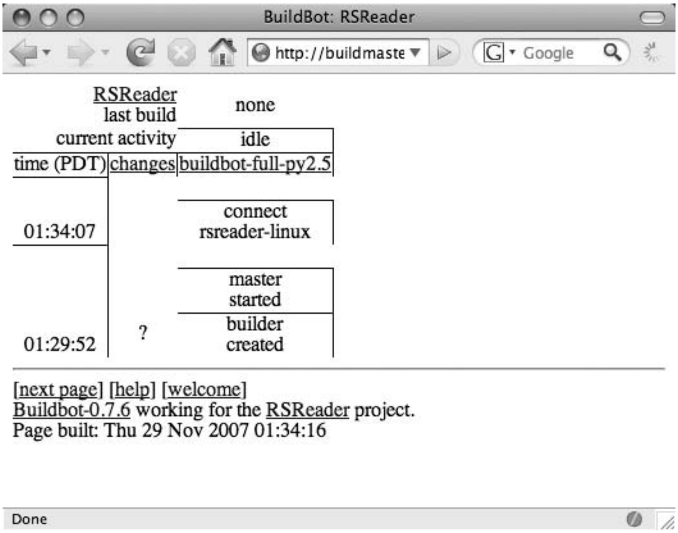


Figure 5-3. *The slave has successfully connected with the master.*

At this point, basic connectivity has been established and a build can be triggered with the command `buildbot sendchange`:

```
$ buildbot sendchange --master buildmaster:4484 -u jeff -n 30 setup.py
```

```
change sent successfully
```

The file name is arbitrary, but the change number specified with `-n` (in this case `30`) is not. The project branch (`rsreader/trunk` in this case) must exist at this revision, or else the build will fail.

The waterfall display immediately shows that the change has been received, and it shows a countdown timer until the build starts. Any changes submitted in this window will reset the timer. This is shown in Figure 5-4.

While the message is being, run the step is rendered in yellow. Once it completes, the step is rendered in green. If the step had failed, it would be red, and if an exception had been encountered, it would be purple. Once the timer expires, the build runs and the slave echoes its message. The output, shown in Figure 5-5, links from the build step.

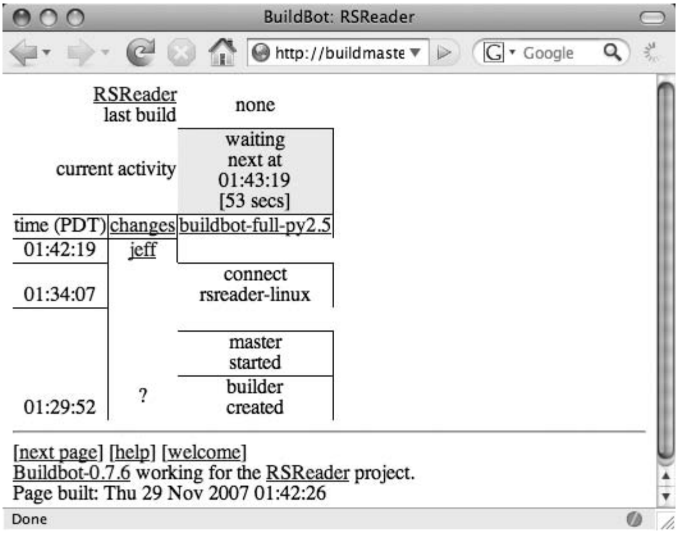


Figure 5-4. *The first build has been triggered.*

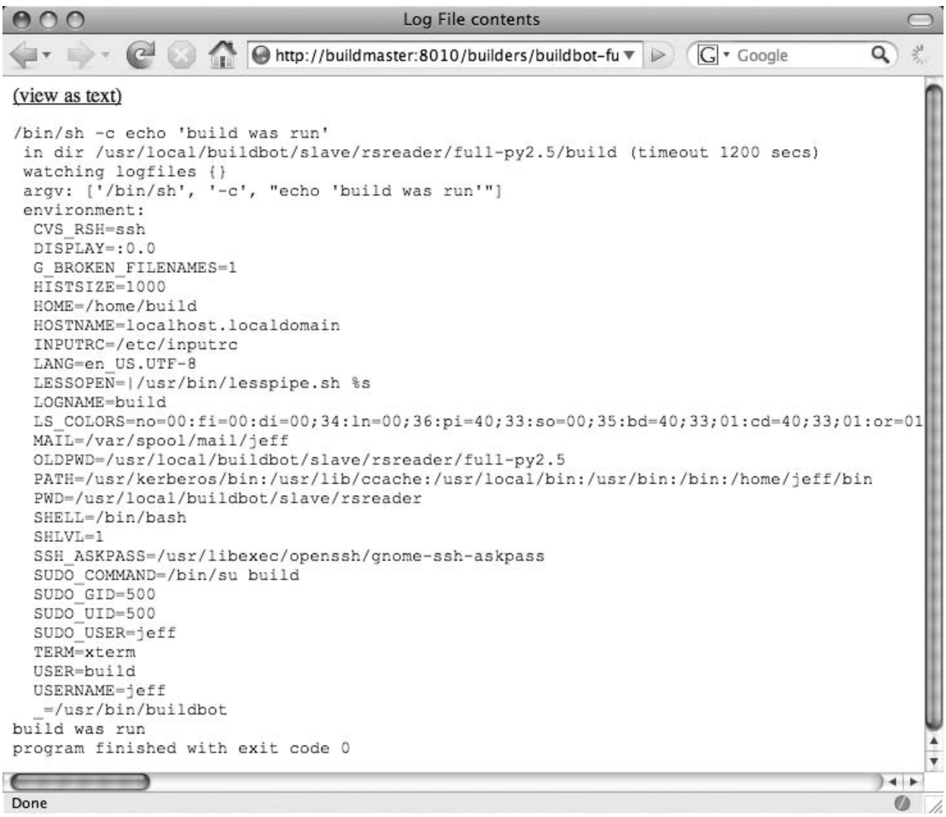


Figure 5-5. *The echo step output*

Information about the step is rendered in blue, and the actual output is rendered in black. It's clear that there is far more information about the step than actual output:

- The command run by the build step is shown in the first line.
- The present working directory follows. It indicates where the command runs from.
- The entire shell environment is displayed. Incorrect environment settings are a common source of build errors, so having this information recorded and available assists with debugging build problems.
- The command's output follows the environment. There's only one line in this case.
- The command's exit code is shown last. As with any UNIX shell command, `0` indicates success.

Hooking Up Source Control

The build master and build slave are now on separate hosts. They both need access to the Subversion repository. Until now, Subversion has been accessed directly through the filesystem, but this will no longer work. However, this isn't a simple choice. Subversion can be accessed remotely via a bewildering spectrum of methods. Repositories can be accessed through the Subversion network server, through WebDAV and the Apache web server, or by tunneling over a shell transport such as SSH. Enumerating the pros and cons of each approach would constitute a chapter's worth of material in itself.

I'm going to choose one pair of methods and stick with them, but if you'd like more information, then I encourage you to consult *Practical Subversion, Second Edition*, by Daniel Berlin and Garrett Rooney (Apress, 2006). The good news is that if you choose another method, then the changes on the client amount to nothing more than changing a URL.

For the rest of the book, I'm choosing a combination of `svnserve` running as a daemon for read-only access and `svnserve` over SSH for write access. This is a common configuration in which anyone can check out files anonymously, but committing changes requires a login and therefore authentication.

Committers need accounts on the Subversion server and write access to the Subversion repository. Authorization is done with group permissions. The repository tree is writable by the Subversion group, and all committers are members of that group. In this book, that group will be named `svn`.

Files created through `svnserve` over SSH are owned by the committer, but they must be writable by the Subversion group. `svnserve` sets the appropriate permissions, but those are affected by the user's umask. The umask turns off selected permissions when files are written. More frequently than not, the user's default umask turns off group write permissions, and it is therefore necessary to override it.

You do this with a wrapper script that replaces `svnserve`. The wrapper script sets the umask to `002` (which turns off writing by others) and calls the original `svnserve` while passing along all the arguments it received:

```
$ sudo mv /usr/local/bin/svnserve /usr/local/bin/svnserve-stock
$ sudo vi /usr/local/bin/svnserve
```

```
... set up the file as below ...
```

```
$ cat /usr/local/bin/svnserve
```

```
#!/bin/sh
umask 002
exec /usr/local/bin/svnserve-stock "$@"
```

```
$ sudo chmod a+x /usr/local/bin/svnserve
```

Now you'll create the user svn and change the ownership of the permissions of the Subversion repository. Remember that the repository is located in /usr/local/svn/repos.

```
$ sudo /usr/sbin/useradd svn
$ sudo chown svn:svn /usr/local/svn/repos
$ sudo chmod g+rw /usr/local/svn/repos
```

You can now start the Subversion server. The --daemon option tells the server to start as a daemon. The -r option tells it where to find the repository. It must be started from the Subversion user's account.

```
$ sudo -u svn /usr/local/bin/svnserve --daemon -r /usr/local/svn/repos
```

The Subversion server is now listening for requests on port 3690. Subversion clients use file:/// URLs to access the local filesystem, and they access svnserve using svn:// URLs. The local URL for the RSReader project is file:///usr/local/svn/repos/rsreader, and it maps to the remote URL svn://source/rsreader. The source component is the host name and the rsreader component is the path relative to the Subversion server's root.

You can verify the status with the svn info command:

```
$ svn info svn://source/rsreader
```

```
Path: rsreader
URL: svn://source/rsreader
Repository Root: svn://source
Repository UUID: e56658fc-2c3c-0410-b453-f6f88bcaf20d
Revision: 30
Node Kind: directory
Last Changed Author: jeff
Last Changed Rev: 30
Last Changed Date: 2007-11-20 13:38:06 -0800 (Tue, 20 Nov 2007)
```

Finally, it's necessary to add committers to the Subversion group. On my system, there are only two users to worry about: jeff and doug. On Linux systems, the usermod command adds users to groups.

```
$ sudo /usr/sbin/usermod -G svn jeff
$ sudo /usr/sbin/usermod -G svn doug
```

The depot is now ready for testing with Subversion over SSH. This access method runs `svnserve` on the repository machine, so it is a local access protocol like the `file:///`. The full directory path is used, yielding `svn+ssh://source/usr/local/svn/repos/rsreader`. Testing this from the command line shows that the URL is valid:

```
$ svn info svn+ssh://source/usr/local/svn/repos/rsreader
```

```
Password:

Path: rsreader
URL: svn+ssh://source/usr/local/svn/repos/rsreader
Repository Root: svn+ssh://source/usr/local/svn/repos
Repository UUID: e56658fc-2c3c-0410-b453-f6f88bcaf20d
Revision: 30
Node Kind: directory
Last Changed Author: jeff
Last Changed Rev: 30
Last Changed Date: 2007-11-20 13:38:06 -0800 (Tue, 20 Nov 2007)
```

Your password is requested before every new connection, but this can be circumvented using SSH keys. Setting up SSH trust relationships isn't very complicated, but it's outside the scope of this book. Tutorials can be found online, and a good one is provided by Linux Journal at `www.linuxjournal.com/article/8759`.

The output indicates that the repository is available, but we know that there is already a copy on the development machine that was checked out from the old file URL. The copy could just be abandoned, but if changes have already been made, then that course of action would be unpalatable. It would be better if there were a way of informing Subversion of the change in locations. The `svn switch` command does just that. The `--relocate` option maps the URLs from one location to another, transforming `file:///` URLs into `svn+ssh://` URLs. That step is performed on the development box, not the Subversion server.

```
$ svn switch --relocate file:/// svn+ssh://source/
```

```
Password:
```

Eclipse automatically recognizes the change in location. The next time Subversion is accessed, Subversive will ask for your credentials, but it may be necessary to open and close the project to get Subversive to correctly display the new URL next to the project.

The final step in setting up Subversion is ensuring that `svnserve` will start when the host machine reboots. As with Buildbot, you do this by putting an appropriate entry in the Subversion user's crontab. On my Subversion server, it looks like this:

```
$ sudo -u svn crontab -l
```

```
@reboot /usr/local/bin/svnserve --daemon -r /usr/local/svn/repos
```

Using the Source

The Subversion repository is now available across the network, so the build slave can now obtain the source code. You add the `SVN` step to the builder factory `f1` in `buildmaster`'s `master.cfg`. Currently, the relevant section reads as follows:

```
from buildbot.process import factory
from buildbot.steps.shell import ShellCommand

f1 = factory.BuildFactory()
f1.addStep(ShellCommand(command="echo 'build was run'"))
```

The `SVN` build step pulls down code from the repository. The `baseURL` points to the Subversion repository. The `baseURL` is concatenated with the branch, so the trailing slash is important. As configured, this branch defaults to `trunk`. The `SVN` step checks out a fresh copy each time when clobber mode is selected.

```
from buildbot.process import factory
from buildbot.steps.source import SVN

f1 = factory.BuildFactory()
f1.addStep(SVN, baseURL="svn://source/rsreader/",
    defaultBranch="trunk",
    mode="clobber",
    timeout=3600)
```

You reconfigure the build master using the command `buildbot reconfig`. This bypasses the need to restart Buildbot. Any error will be reported in `twistd.log`, and Buildbot will continue running with the old configuration.

```
$ sudo -u build buildbot reconfig /usr/local/buildbot/master/rsreader
```

```
sending SIGHUP to process 2152
2008-05-05 15:59:56-0700 [-] loading configuration from /usr/local/buildbot/master
…
Reconfiguration appears to have completed successfully.
```

The reconfiguration is reflected in the waterfall display too. This is shown in Figure 5-6.

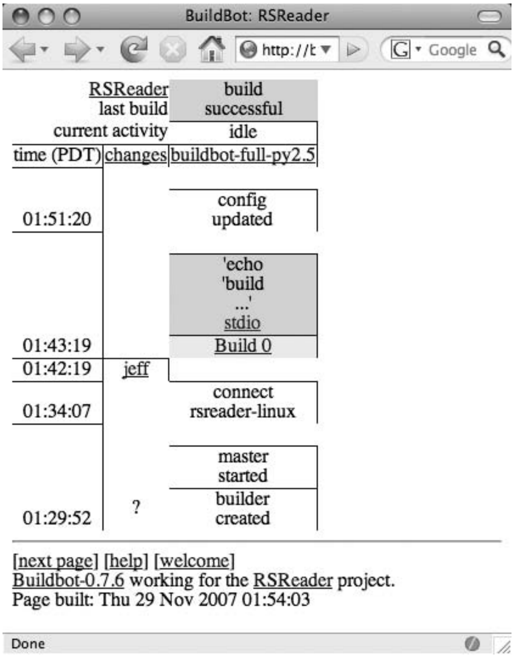


Figure 5-6. *Successful reconfiguration*

If all is configured correctly, then the next build will retrieve the source code from the repository. The build is again triggered using `buildbot sendchange`, and the waterfall display is monitored. If everything worked, then the `SVN` step will appear, and once it completes, it will be green, as shown in Figure 5-7.

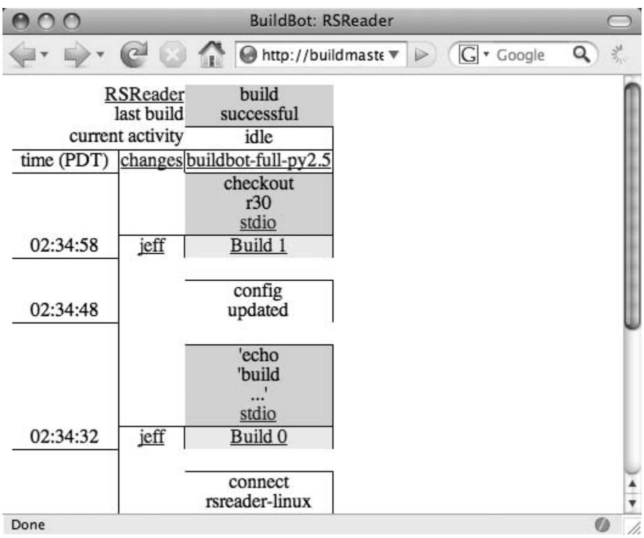


Figure 5-7. *The slave successfully checks out code.*

The successful checkout indicates that the reconfiguration was successful, that all the parameters are correct, and that the build slave can retrieve code from Subversion. It is useful to note that the checkout message in the `SVN` step includes the revision number; in this case revision 30 (`r30`).

Subversion to Buildbot, Over

Subversion should send notifications to Buildbot when changes are committed. This is done with hooks. *Hooks* are programs triggered by events in Subversion. The events are commits, locking, and revision property changes, and there are hooks for several steps in each kind of event. The commit event is the interesting one for our purpose.

The commit process has three hooks. The *start-commit* hook is called before the commit transaction is created. The *pre-commit* hook is called after the commit transaction has been created, but before it has been submitted. Both of these hooks can abort the commit. The *post-commit* hook is called after a successful commit, and it is the one of interest. It takes the repository path and the created revision as arguments, and its return code is ignored.

The hook sends notifications using `svn_buildbot.py`. This program ships in the Buildbot `contrib` directory. Recall that you installed Buildbot with `easy_install -b /tmp/bbinst buildbot`, and that left a copy of the full package in `/tmp/bbinst`. You can copy `svn_buildbot.py` from there to the Subversion directories.

The hooks themselves are stored in the directory `/usr/local/svn/repos/hooks`. As shipped, the directory contains templates demonstrating how each hook is used. The post-commit hook is named `/usr/local/svn/repos/hooks/post-commit`, and it must be executable.

```
$ sudo -u svn mkdir /usr/local/svn/bin
$ sudo -u svn cp /tmp/bbinst/buildbot/contrib/svn_buildbot.py➥
 /usr/local/svn/bin/
$ sudo -u svn vi /usr/local/svn/repos/hooks/post-commit
```

```
...some editing...
```

```
$ sudo -u svn chmod a+x /usr/local/svn/repos/hooks/post-commit
$ cat /usr/local/svn/repos/hooks/post-commit
```

```
#!/bin/sh
REPOS="$1"
REV="$2"
MASTER=buildmaster
PORT=4484
/usr/local/svn/bin/svn_buildbot.py --repository "$REPOS" \
                                   --revision "$REV" \
                                   --bbserver $MASTER \
                                   --bbport $PORT
```

The final step is testing the hook by submitting a change to the codeline and then checking the result on the waterfall display, as shown in Figure 5-8. This is done on the development machine:

```
$ cat " " >>  setup.py
$ svn commit -m "Just a minor change to trigger a build"
```

```
Password:

Sending        setup.py
Transmitting file data .
Committed revision 31.
```

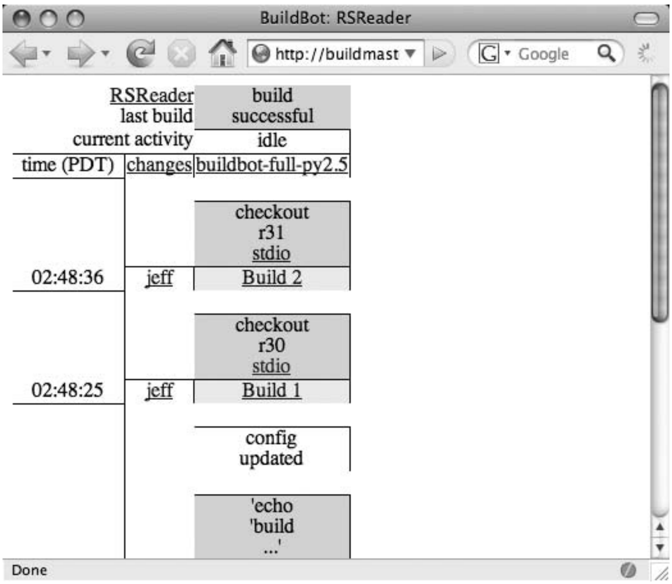


Figure 5-8. *The build was successfully triggered by a Subversion submission.*

A Python for Every Builder

I haven't said this for a while, so I'll say it again. The goal is to produce a clean build every time. This requires removing all packages and installed scripts from the Python installation. The easiest way of preventing builders from stepping on each other is to provide each one with its own interpreter. Some people may disagree with me, but disk space is cheap, and the cleansing process is straightforward and easily automated.

Python is installed into the build slave's root directory. The Python version is explicitly named so that multiple Python versions can be installed in the same build slave. In this case, the Python build prefix will be `/usr/local/buildbot/slave/rsreader/full-py2.5/python2.5`.

The decision not to track the minor version is a conscious one. If there comes a point where the minor Python revisions are important, then I will track them.

```
$ curl -L -o Python-2.5.1.tgz http://www.python.org/ftp/➥
python/2.5.1/Python-2.5.1.tgz
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 10.5M  100 10.5M    0     0  31388      0  0:05:52  0:05:52 --:--:-- 49899
```

```
$ tar xvfz Python-2.5.1.tgz
```

```
Python-2.5.1/
Python-2.5.1/Python/
...
Python-2.5.1/pyconfig.h.in
Python-2.5.1/install-sh
```

```
$ cd Python-2.5.1
$ ./configure --prefix=/usr/local/buildbot/slave/rsreader/full-py2.5/python2.5
```

```
checking MACHDEP... linux2
checking EXTRAPLATDIR...
... many minutes pass ...
creating Modules/Setup.local
creating Makefile
```

```
$ make
```

```
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall➥
-Wstrict-prototypes  -I. -I./Include   -DPy_BUILD_CORE➥
 -o Modules/python.o ./Modules/python.c
... many more minutes pass ...
changing mode of build/scripts-2.5/idle from 664 to 775
changing mode of build/scripts-2.5/smtpd.py from 664 to 775
```

```
$ make test
```

```
case $MAKEFLAGS in \
        *-s*)  CC='gcc -pthread' LDSHARED='gcc -pthread -shared'➥
 OPT='-DNDEBUG -g -O3 -Wall -Wstrict-prototypes' ./python -E
...
    test_timeout test_urllib2net test_urllibnet test_winreg
    test_winsound test_zipfile64
```

```
$ make install
```

```
/usr/bin/install -c python /usr/local/buildbot/slave/rsreader/full-py2.5/➥
python2.5/bin/python2.5
if test -f libpython2.5.so; then \
...
/usr/bin/install -c -m 644 ./Misc/python.man \
                /usr/local/buildbot/slave/rsreader/python2.5/man/man1/python.1
```

Finally, a Real Build Succeeds

Builds are produced using the `Compile` step. The `Compile` step will run the statement `python setup.py build` as if run from the command line. However, it should use the private Python interpreter installed in the previous section. Absolute paths are out of the question. The build clients may be rearranged in the future, or they may be relocated by others with good reasons for placing them elsewhere, so relative paths should be used. There are many paths to keep straight, so they're summarized in Table 5-1.

Table 5-1. *Paths Used on the Slave*

Path	Description
`/usr/local/buildbot/slave/rsreader`	The build slave's root directory
`/usr/local/buildbot/slave/rsreader/full-py2.5`	The *builder* directory defined in `master.cfg`
`/usr/local/buildbot/slave/rsreader/full-py2.5/build`	The *build* directory where the builder factory runs
`/usr/local/buildbot/slave/rsreader/full-py2.5/python2.5`	The slave's local Python 2.5 installation
`/usr/.../full-py2.5/python2.5/bin/python`	The Python interpreter
`/usr/.../full-py2.5/python2.5/lib/python2.5/site-packages`	Locally installed packages
`/usr/.../full-py2.5/python2.5/site-bin`	Locally installed executables
`../python2.5/bin/python`	The relative path from the build directory to the interpreter
`../python2.5/lib/python2.5/site-packages`	The relative path from the build directory to the locally installed packages
`../python2.5/site-bin`	The relative path from the build directory to the locally installed executables

The `SVN` step checks out the code into the directory `full-py2.5/build` relative to the slave's directory. The builder directory, `full-py2.5`, is specified in the builder's definition in `master.cfg`, and the last subdirectory is always `build`. The builder executes in this directory. The relative path from the build directory to the locally installed Python 2.5 interpreter is `../python2.5/bin/python`.

After adding the new step, the relevant section of `master.cfg` looks like this:

```
from buildbot.process import factory
from buildbot.steps.source import SVN
from buildbot.steps.shell import Compile

f1 = factory.BuildFactory()
f1.addStep(SVN, baseURL="svn://repos/rsreader/",
    defaultBranch="trunk",
    mode="clobber",
    timeout=3600)
f1.addStep(Compile, command=["../python2.5/bin/python",
                             "./setup.py",
                             "build"])
```

You reconfigure Buildbot and trigger a build with `buildbot sendchange`, and the change is reflected in the waterfall display. Figure 5-9 shows the completed build.

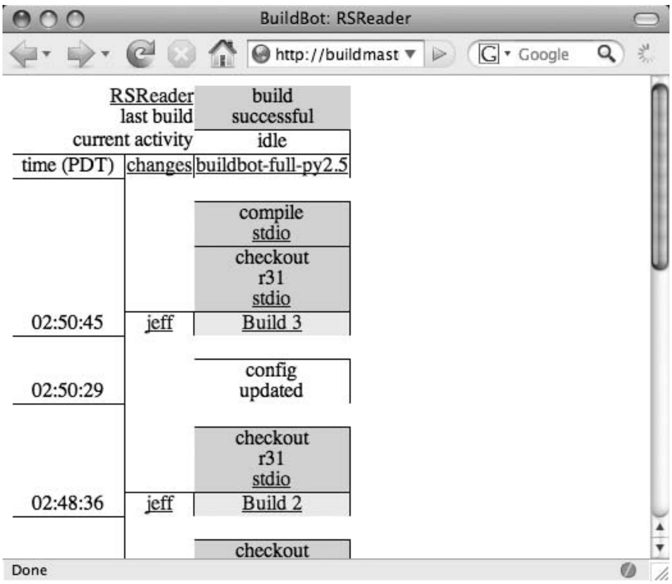


Figure 5-9. *The build step succeeds.*

Installing the Build

The `Install` step will generate executables. By default, these executables will be placed into the local Python `bin` directory along with the Python interpreter and other stock tools. The build will need to remove the generated artifacts—however, separating them from the preexisting tools is problematic. Fortunately, you can specify a different directory for executables with the `--install-script` option. This is not an issue for packages, as they are installed into `site-packages`. It contains no stock Python files, so it can be cleansed with impunity.

```
from buildbot.process import factory
from buildbot.steps.source import SVN
from buildbot.steps.shell import Compile, ShellCommand

f1 = factory.BuildFactory()
f1.addStep(SVN, baseURL="svn://repos/rsreader/",
    defaultBranch="trunk",
    mode="clobber",
    timeout=3600)
f1.addStep(ShellCommand, command=["mkdir",
                                  "../python2.5/site-bin"])
f1.addStep(Compile, command=["../python2.5/bin/python",
                             "./setup.py",
                             "build"])
f1.addStep(Compile, command=["../python2.5/bin/python",
                             "./setup.py",
                             "install",
                             "--install-scripts",
                             "../python2.5/site-bin"])
```

The `Install` step output shown in Figure 5-9 indicates that the `docutils` package was installed. Triggering the build a second time yields the log shown in Figure 5-10. It shows that `docutils` was not actually installed. Instead, the previous installation was used. This may seem a pedantic point, but I've encountered many situations in which a clean install would fail for one reason or another, but subsequent installations would succeed. It's not an acceptable answer to simply tell your customer, "Just reinstall it, and it will work." Each build should yield a clean result.

The build code functions, but it is getting messy. The build code is going to be around as long as the application—perhaps even longer. There is a tendency to neglect build configurations and build code. Normal programming practices aren't applied, and eventually the code rots under the weight of neglect. Changes to build code are easy to make if they're small, and the key to keeping them small is making the changes as the need is recognized.

Both the path to `python` and the path to the `site-bin` directory are replicated. We'll extract them into constants. This is refactoring—changing the structure of code to improve readability and maintainability without altering its function. Refactoring is best done when the code can be tested. Fortunately, the build configuration code has a built-in test and test harness, which is the build system itself. If you can't make a build, then your configuration changes are broken.

```
from buildbot.process import factory
from buildbot.steps.source import SVN
from buildbot.steps.shell import Compile, Install, ShellCommand

python = "../python2.5/bin/python"
site_bin = "../python2.5/site-bin"
```

```
f1 = factory.BuildFactory()
f1.addStep(SVN, baseURL="svn://repos/rsreader/",
    defaultBranch="trunk",
    mode="clobber",
    timeout=3600)
f1.addStep(ShellCommand, command=["mkdir", build_bin])
f1.addStep(Compile, command=[python, "./setup.py", "build"])
f1.addStep(Compile, command=[python, "./setup.py",  "install",
                             "--install-scripts", site-bin])
```

Here, we're saving master.cfg, reconfiguring Buildbot, and triggering a build. The results are the same, which is good and bad. It's good because we've verified your changes, and they work as expected. It's bad because the old build is still installed, so the installation directories python2.5/lib/2.5/site-packages and python2.5/site-bin must be removed and recreated.

```
from buildbot.process import factory
from buildbot.steps.source import SVN
from buildbot.steps.shell import Compile, ShellCommand

python = "../python2.5/bin/python"
site_bin = "../python2.5/site-bin"
site_pkgs = "../python2.5/lib/python2.5/site-packages"

f1 = factory.BuildFactory()
f1.addStep(SVN, baseURL="svn://repos/rsreader/",
    defaultBranch="trunk
    mode="clobber",
    timeout=3600)
f1.addStep(ShellCommand, command=["rm", "-rf", site_pkgs])
f1.addStep(ShellCommand, command=["mkdir", site_pkgs])
f1.addStep(ShellCommand, command=["rm", "-rf", site_bin])
f1.addStep(ShellCommand, command=["mkdir", site_bin])
f1.addStep(Compile, command=[python, "./setup.py", "build"])
f1.addStep(Compile, command=[python, "./setup.py",  "install",
                             "--install-scripts", site_bin])
```

Once again, we're saving the changes, reconfiguring Buildbot, and triggering a build. The waterfall display is shown in Figure 5-10. It clearly shows the new step and the last step's output—it is clear that the packages have been freshly installed.

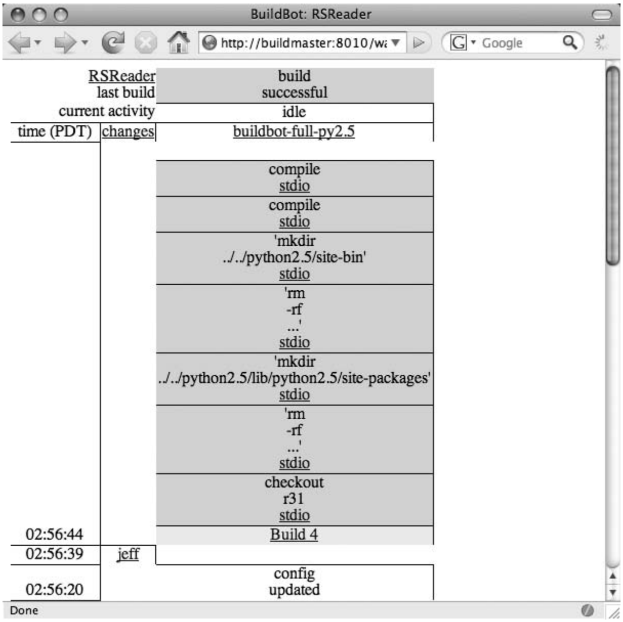


Figure 5-10. *A complete clean build*

Supporting Python 2.4 Builds

This chapter's ultimate goal is supporting both Python 2.4 and Python 2.5, and we're getting very close. Python 2.4 must be installed, a new scheduler added, and a new builder defined. The master configuration must also be refactored along the way.

Installing Python 2.4 is precisely analogous to installing Python 2.5. It is placed in a directory named `python2.4`, directly beneath the slave's root directory. Supplying the new directory to the `./configure` step is the only change in procedure.

```
$ curl -L -o Python-2.4.4.tgz http://www.python.org/ftp/python/2.4.4/➥
Python-2.4.4.tgz
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 10.5M  100 10.5M    0     0  31388      0  0:05:52  0:05:52 --:--:-- 49899
```

```
$ tar xvfz Python-2.4.4.tgz
```

```
Python-2.4.4/
Python-2.4.4/Python/
...
Python-2.4.4/pyconfig.h.in
Python-2.4.4/install-sh
```

```
$ cd Python-2.4.4
$ ./configure --prefix=/usr/local/buildbot/slave/rsreader/full-py2.4/python2.4
```

```
checking MACHDEP... linux2
checking EXTRAPLATDIR...
... many minutes pass ...
creating Modules/Setup.local
creating Makefile
```

```
$ make; make test; make install
```

```
gcc -pthread -c -fno-strict-aliasing -DNDEBUG -g -O3 -Wall➥
-Wstrict-prototypes  -I. -I./Include   -DPy_BUILD_CORE➥
 -o Modules/python.o ./Modules/python.c
... many more minutes pass ...
changing mode of build/scripts-2.4/idle from 664 to 775
changing mode of build/scripts-2.4/smtpd.py from 664 to 775
```

There are three things that must be done to a new builder. First, the factory producing it must be created. Then it must be created using that factory. Finally, the builder has to be scheduled. In this case, the process amounts to little more than duplicating the Python 2.5 definitions and changing the version number to 2.4, although some new constants will need to be created along the way.

The clarity-to-maintainability ratio for the scheduler and builder sections clearly favors duplication. Just as clearly, the clarity-to-maintainability ratio militates against duplicating the builder factory definition. It doubles the number of constants and the number of lines. If the build process is modified, it will need to be modified in both places, and I guarantee it will be modified before the chapter is out. There is much to be gained from refactoring here.

You'll encapsulate the builder factory in a function. That function will take the Python version as its argument, and it will return a builder factory for that Python version. Along the way, you'll extract many of the constants into functions.

The changes are made in two parts. You'll refactor the builder and builder factory and test them. Then the new 2.4 builder and schedulers will be added. This isolates the changes in each step, making debugging much easier.

```
####### BUILDERS

from buildbot.process import factory
from buildbot.steps.source import SVN
from buildbot.steps.shell import Compile, ShellCommand

def python_(version):
    return  "../python%s/bin/python" % version

def site_bin_(version):
    return  "../python%s/site-bin" % version
```

```
def site_pkgs_(version):
    subst = {'v': version}
    path = "../python%(v)s/lib/python%(v)s/site-packages"
    return path % subst

def pythonBuilder(version):
    python = python_(version)
    site_bin = site_bin_(version)
    site_pkgs = site_pkgs_(version)

    f = factory.BuildFactory()
    f.addStep(SVN, baseURL="svn://repos/rsreader/",
              defaultBranch="trunk",
              mode="clobber",
              timeout=3600)
    f.addStep(ShellCommand, command=["rm", "-rf", site_pkgs])
    f.addStep(ShellCommand, command=["mkdir", site_pkgs])
    f.addStep(ShellCommand, command=["rm", "-rf", site_bin])
    f.addStep(ShellCommand, command=["mkdir", site_bin])
    f.addStep(Compile, command=[python, "./setup.py", "build"])
    f.addStep(Compile, command=[python, "./setup.py", "install",
                                "--install-scripts", site_bin])
    return f

b1 = {'name': "buildbot-full-py2.5",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.5",
       'factory': pythonBuilder('2.5'),
      }
```

You've now parameterized the builder factory. A reconfiguration and rebuild verifies that it works correctly. Now that you've made these changes, the Python 2.4 builder can be added. You'll add the schedule, define the builder, and add the builder to the `builders` property:

```
####### SCHEDULERS

from buildbot.scheduler import Scheduler
c['schedulers'] = []
c['schedulers'].append(Scheduler(name="rsreader under python 2.5",
                       branch=None,
                       treeStableTimer=60,
                       builderNames=["buildbot-full-py2.5"]))
c['schedulers'].append(Scheduler(name="rsreader under python 2.4",
                       branch=None,
                       treeStableTimer=60,
                       builderNames=["buildbot-full-py2.4"]))
```

```
...

####### BUILDERS

...

b1 = {'name': "buildbot-full-py2.5",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.5",
      'factory': pythonBuilder('2.5'),
      }
b2 = {'name': "buildbot-full-py2.4",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.4",
      'factory': pythonBuilder('2.4'),
      }

c['builders'] = [b1, b2]
```

This time when you run the build, the second builder shows up in a second column, as in Figure 5-11.

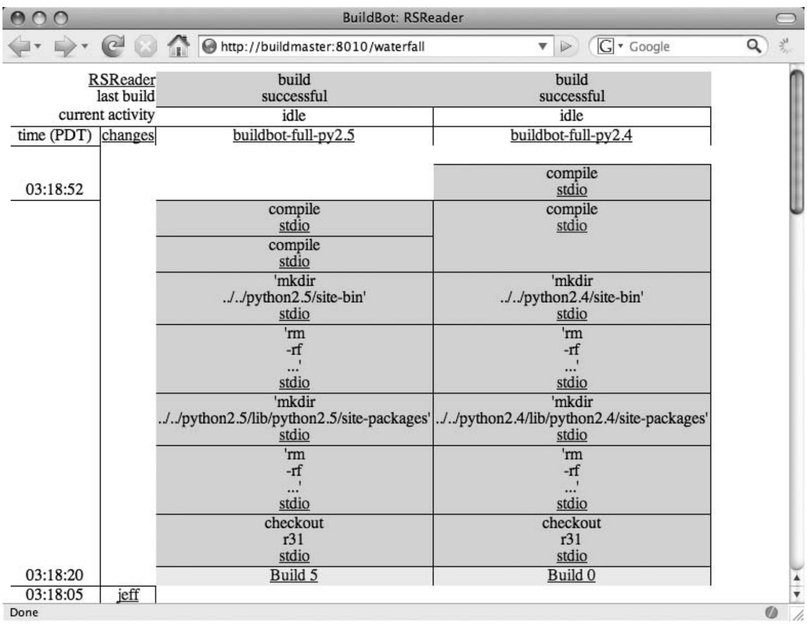


Figure 5-11. *Simultaneous Python 2.4 and 2.5 builds*

The second builder executes in parallel with the first. There are differences that are immediately apparent. The build step takes much longer under 2.4 than under 2.5. If you look at the log, the reason for this should quickly become clear: `ez_setup.py` is not using the locally provided copy of Setuptools; it is downloading Setuptools from the network instead.

Ensuring Local Dependency Processing

In Chapter 6, which deals with unit testing, I introduce a package called Nose. It will be a required dependency for RSReader. Adding it now gives me an opportunity to demonstrate how to restrict dependencies to local installation. This is done through `easy_install`'s `--allow-hosts` option. If the option is defined, then `easy_install` will only download eggs from servers whose host name matches its pattern. No hosts are matched if the pattern is `"None"`, so this effectively blocks all external access.

The Setuptools `install` command calls `easy_install` to process missing dependencies. It is the `easy_install` command that observes the `allow-hosts` option. Unfortunately, `install` knows nothing about the `--allow-hosts` option, and there is no way to hand the option directly from `install` to `easy_install`. However, it can be specified in the project's `setup.cfg` file.

The build server should always enforce this option to catch missing packages. It could be set in `setup.cfg` within the codeline—and indeed it may always be—but if it is removed, then the install will silently retrieve the packages from the network. Instead, we'll use the command `setup.py setopt` to fix the value before each build begins:

```
f = factory.BuildFactory()
f.addStep(SVN, baseURL="svn://repos/rsreader/",
          defaultBranch="trunk",
          mode="clobber",
          timeout=3600)
f.addStep(ShellCommand, command=["rm", "-rf", site_pkgs])
f.addStep(ShellCommand, command=["mkdir", site_pkgs])
f.addStep(ShellCommand, command=["rm", "-rf", site_bin])
f.addStep(ShellCommand, command=["mkdir", site_bin])
f.addStep(ShellCommand,
          command=[python, "./setup.py", "setopt",
                   ""--command", "easy_install",
                     "--option", "allow-hosts",
                     "--set-value", "None"])
f.addStep(Compile, command=[python, "./setup.py", "build"])
f.addStep(Compile, command=[python, "./setup.py", "install",
                            "--install-scripts", site_bin])
```

As always, when you make a change, you should perform a test build. Remember that the goal is ensuring build failures when a required package is missing. To check this, you add a requirement to the project's `setup.py` file without supplying the package in the project's `thirdparty` directory.

```
install_requires = [
                     'docutils == 0.4',
                     'nose == 0.10.0',
                     ],
```

You commit this change to Subversion, and it automatically triggers a build. The installation step fails, as in Figure 5-12. The failed step's output indicates that the package could not be located locally. As hoped, a missing package results in a build failure.

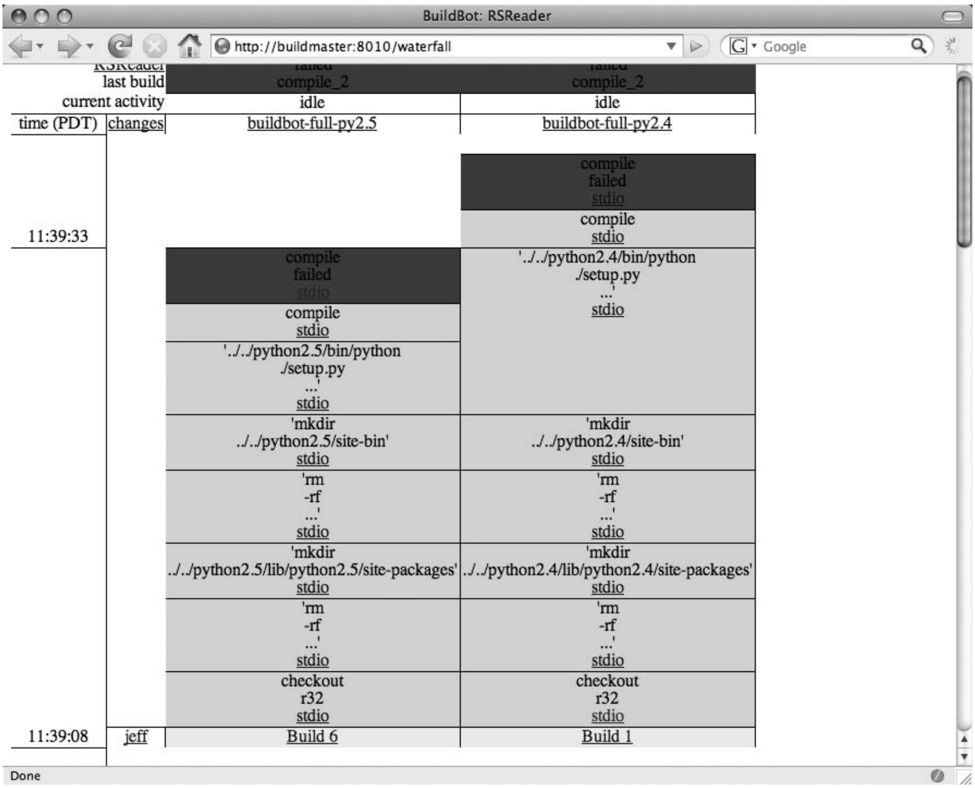


Figure 5-12. *The Install step fails when a dependency is missing.*

You download nose from `http://somethingaboutorange.com/mrl/projects/nose/nose-0.10.0.tar.gz`, and check the compressed archive into the `thirdparty` directory. You commit the change, and a build happens automatically. This time the build succeeds, as shown in Figure 5-13.

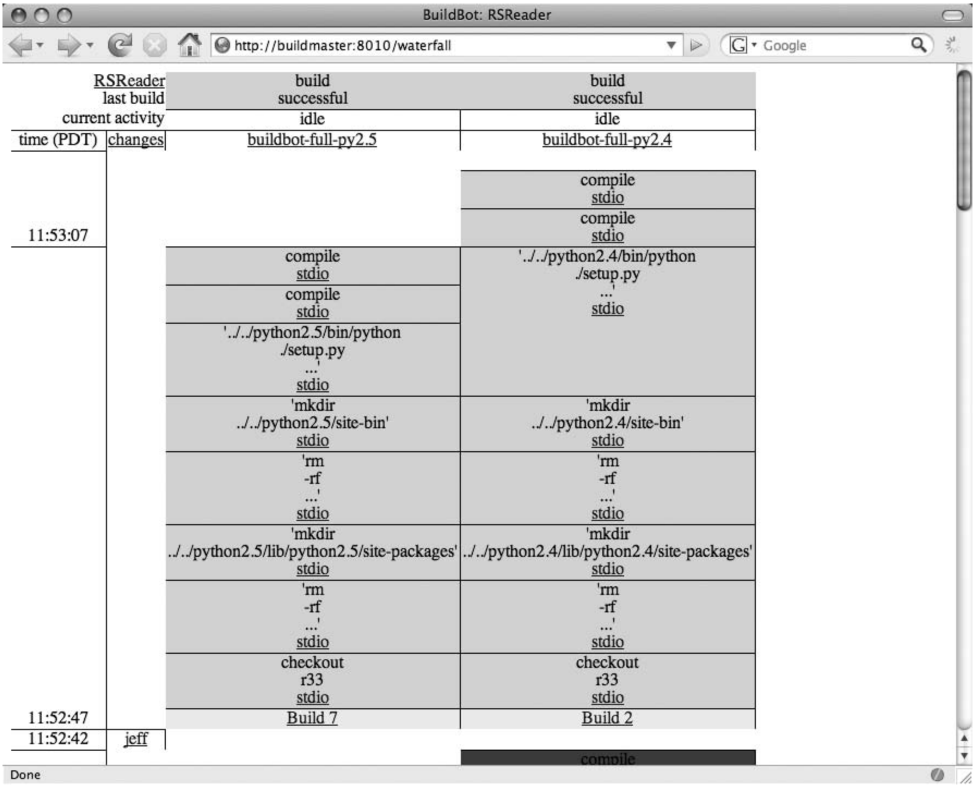


Figure 5-13. *The Install step succeeds when the dependency package is added.*

Keeping Up Appearances

The waterfall display records the name of each step in gory detail. That detailed information is available from each step's output. The information presented in the waterfall display should be understandable at a glance. The only step with an immediately clear meaning is `Compile`. The description presented for each step should be equally succinct and precise.

You accomplish this by modifying a pair of keyword properties in each build step. The message contained in the `description` keyword is shown while a step is in progress, and the message contained in the `descriptionDone` keyword is shown when a step is complete. We can add these keywords to each step to increase the waterfall display's clarity:

```
f = factory.BuildFactory()
f.addStep(SVN, baseURL="svn://repos/rsreader/",
   defaultBranch="trunk",
   mode="clobber",
   timeout=3600)
f.addStep(ShellCommand,
          command=["rm", "-rf", site_pkgs],
          description="removing old site-packages",
          descriptionDone="site-packages removed")
f.addStep(ShellCommand,
          command=["mkdir", site_pkgs],
          description="creating new site-packages",
          descriptionDone="site-packages created")
f.addStep(ShellCommand,
          command=["rm", "-rf", site_bin],
          description="removing old site-bin",
          description="site-bin removed")
f.addStep(ShellCommand,
          command=["mkdir", site_bin],
          description="creating new site-bin",
          descriptionDone="site-bin created")
f.addStep(ShellCommand,
          command=[python, "./setup.py", "setopt",
                   ""--command", "easy_install",
                     "--option", "allow-hosts",
                     "--set-value", "None"],
          description="Setting allow-hosts to None",
          descriptionDone="Allow-hosts set to None")
f.addStep(Compile, command=[python, "./setup.py", "build"])
f.addStep(ShellCommand,
          command=[python, "./setup.py",  "install",
                   "--install-scripts", site_bin],
          description="Installing",
          descriptionDone="Installed")
return f
```

The resulting waterfall display is shown in Figure 5-14. The labels are more concise and informative, and the resulting display uses less space. When using many builders, this can become a significant factor. It gives me a warm, fuzzy feeling, too, and that counts for a lot.

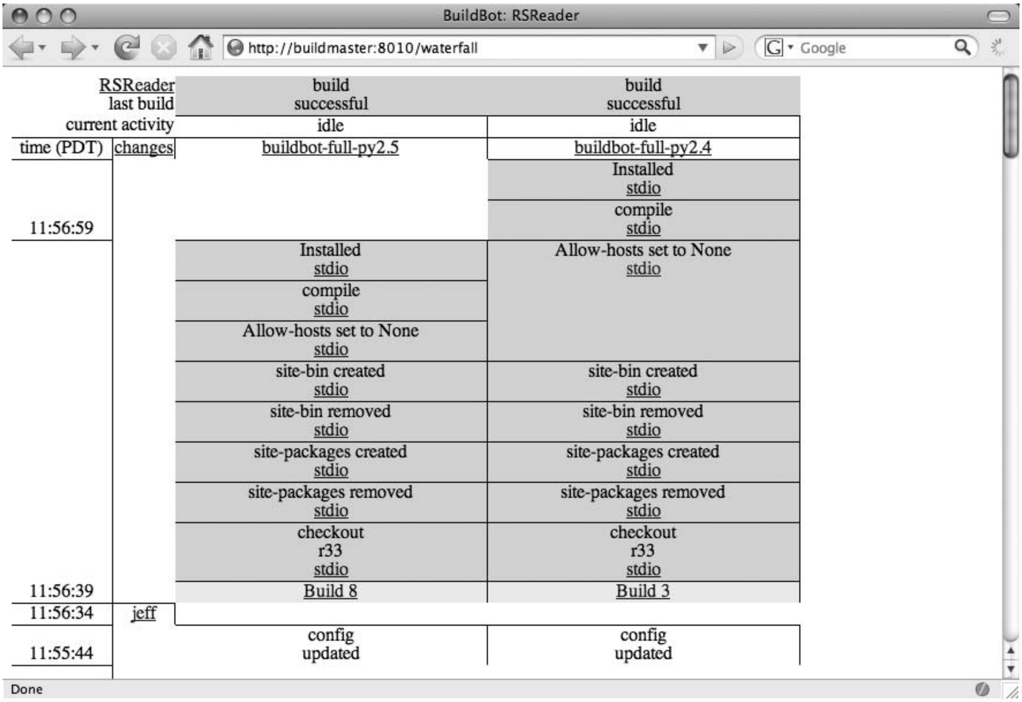


Figure 5-14. *More readable step descriptions*

Summary

Clean, repeatable builds are an easily achievable outcome using an external build system. Buildbot is one such system. A Buildbot system consists of four components: the Subversion server, the Buildbot master, one or more Buildbot slaves, and the development environments. A remotely accessible Subversion repository allows these roles to be distributed to many hosts.

Buildbot is implemented on top of Twisted. Buildbot and Twisted must be installed on the Buildbot master, the Buildbot slaves, and the Subversion repository server. It is not necessary to install it on the developers' machines. The Subversion repository triggers builds whenever code is committed. This is done through a post-commit hook.

The *build master* controls and configures *build slaves*, which are the machines that perform builds. The Buildbot master finds out about unprocessed changes through *change sources*. S*chedulers* trigger builders when certain conditions are satisfied. A *builder* ties together a *builder factory* and a build slave, and it defines a directory where the build will be performed. Builder factories generate the steps to perform a build, and a *build step* is an action that performs one step of a build.

Build steps include actions such as synching source from a repository, compiling an application, and installing the compiled application.

The system I discussed uses per-builder Python installations. These allow the build system to completely remove all installed packages and executables when a new build is performed, and this is done without impacting the rest of the system. This allows the build to verify that completely clean installations are self-contained.

Buildbot has additional capabilities you can configure. Among these is the ability to run unit tests for each build. I haven't demonstrated this yet, but I will in the next chapter, which covers the basics of unit testing.

CHAPTER 6

Testing: The Horse and the Cart

This chapter describes unit testing and test-driven development (TDD); it focuses primarily on the infrastructure supporting those practices. I'll expose you to the practices themselves, but only to the extent necessary to appreciate the infrastructure. Along the way, I'll introduce the crudest flavors of agile design, and lead you through the development of a set of acceptance tests for the RSReader application introduced in Chapter 5. This lays the groundwork for Chapter 7, where we'll explore the TDD process and the individual techniques involved.

All of this begs the question, "What are unit tests?" Unit tests verify the behavior of small sections of a program in isolation from the assembled system. Unit tests fall into two broad categories: programmer tests and customer tests. What they test distinguishes them from each other.

Programmer tests prove that the code does what the programmer expects it to do. They verify that the code works. They typically verify behavior of individual methods in isolation, and they peer deeply into the mechanisms of the code. They are used solely by developers, and they are not be confused with customer tests.

Customer tests (a.k.a. acceptance tests) prove that the code behaves as the customer expects. They verify that the code works correctly. They typically verify behavior at the level of classes and complete interfaces. They don't generally specify *how* results are obtained; they instead focus on *what* results are obtained. They are not necessarily written by programmers, and they are used by everyone in the development chain. Developers use them to verify that they are building the right thing, and customers use them to verify that the right thing was built.

In a perfect world, specifications would be received as customer tests. Alas, this doesn't happen often in our imperfect world. Instead, developers are called upon to flesh out the design of the program in conjunction with the customer. Designs are received as only the coarsest of descriptions, and a conversation is carried out, resulting in detailed information that is used to formulate customer tests.

Unit testing can be contrasted with other kinds of testing. Those other kinds fall into the categories of functional testing and performance testing.

Functional testing verifies that the complete application behaves as expected. Functional testing is usually performed by the QA department. In an agile environment, the QA process is directly integrated into the development process. It verifies what the customer sees, and it examines bugs resulting from emergent behaviors, real-life data sets, or long runtimes.

Functional tests are concerned with the internal construction of an application only to the extent that it impinges upon application-level behaviors. Testers don't care if the application was written using an array of drunken monkeys typing on IBM Selectric typewriters run through a bank of badly tuned analog synthesizers before finally being dumped into the source repository. Indeed, some testers might argue that this process would produce better results.

Functional testing falls into four broad categories: exploratory testing, acceptance testing, integration testing, and performance testing. *Exploratory testing* looks for new bugs. It's an inventive and sadistic discipline that requires a creative mindset and deep wells of pessimism. Sometimes it involves testers pounding the application until they find some unanticipated situation that reveals an unnoticed bug. Sometimes it involves locating and reproducing bugs reported from the field. It is an interactive process of discovery that terminates with test cases characterizing the discovered bugs.

Acceptance testing verifies that the program meets the customer's expectations. Acceptance tests are written in conjunction with the customer, with the customer supplying the domain-specific knowledge, and the developers supplying a concrete implementation. In the best cases, they supplant formal requirements, technical design documents, and testing plans. They will be covered in detail in Chapter 11.

Integration testing verifies that the components of the system interact correctly when they are combined. Integration testing is not necessarily an end-to-end test of the application, but instead verifies blocks larger than a single unit. The tools and techniques borrow heavily from both unit testing and acceptance testing, and many tests in both acceptance and unit test suites can often be characterized as integration tests.

Regression testing verifies that bugs previously discovered by exploratory testing have been fixed, or that they have not been reintroduced. The regression tests themselves are the products of exploratory testing. Regression testing is generally automated. The test coverage is extensive, and the whole test suite is run against builds on a frequent basis.

Performance testing is the other broad category of functional testing. It looks at the overall resource utilization of a live system, and it looks at interactions with deployed resources. It's done with a stable system that resembles a production environment as closely as possible.

Performance testing is an umbrella term encompassing three different but closely related kinds of testing. The first is what performance testers themselves refer to as performance testing. The two other kinds are stress testing and load testing. The goal of performance testing is not to find bugs, but to find and eliminate bottlenecks. It also establishes a baseline for future regression testing.

Load testing pushes a system to its limits. Extreme but expected loads are fed to the system. It is made to operate for long periods of time, and performance is observed. Load testing is also called volume testing or endurance testing. The goal is not to break the system, but to see how it responds under extreme conditions.

Stress testing pushes a system beyond its limits. Stress testing seeks to overwhelm the system by feeding it absurdly large tasks or by disabling portions of the system. A 50 GB e-mail attachment may be sent to a system with only 25 GB of storage, or the database may be shut down in the middle of a transaction. There is a method to this madness: ensuring recoverability. Recoverable systems fail and recover gracefully rather than keeling over disastrously. This characteristic is important in online systems.

Sadly, performance testing isn't within this book's scope. Functional testing, and specifically acceptance testing, will be given its due in Chapter 11.

Unit Testing

The focus in this chapter is on programmer tests. From this point forward, I shall use the terms *unit test* and *programmer test* interchangeably. If I need to refer to customer tests, I'll name them explicitly.

So why unit testing? Simply put, unit testing makes your life easier. You'll spend less time debugging and documenting, and it results in better designs. These are broad claims, so I'll spend some time backing them up.

Developers resort to debugging when a bug's location can't be easily deduced. Extensive unit tests exercise components of the system separately. This catches many bugs that would otherwise appear once the lower layers of a system are called by higher layers. The tests rigorously exercise the capabilities of a code module, and at the same time operate at a fine granularity to expose the location of a bug without resorting to a debugger.

This does not mean that debuggers are useless or superfluous, but that they are used less frequently and in fewer situations. Debuggers become an exploratory tool for creating missing unit tests, and for locating integration defects.

Unit tests document intent by specifying a method's inputs and outputs. They specify the exceptional cases and expected behaviors, and they outline how each method interacts with the rest of the system. As long as the tests are kept up to date, they will always match the software they purport to describe. Unlike other forms of documentation, this coherence can be verified through automation.

Perhaps the most far-fetched claim is that unit tests improve software designs. Most programmers can recognize a good design when they see it, although they may not be able to articulate why it is good. What makes a good design? Good designs are highly cohesive and loosely coupled.

Cohesion attempts to measure how tightly focused a software module is. A module in which each function or method focuses on completing part of a single task, and in which the module as a whole performs a single well-defined task on closely related sets of data, is said to be highly cohesive. High cohesion promotes encapsulation, but it often results in high coupling between methods.

Coupling concerns the connections between modules. In a loosely coupled system, there are few interactions between modules, with each depending only on a few other modules. The points where these dependencies are introduced are often explicit. Instead of being hardcoded, objects are passed into methods and functions. This limits the "ripple effect" where changes to one module result in changes to many other modules.

Unit testing improves designs by making the costs of bad design explicit to the programmer as the software is written. Complicated software with low cohesion and tight coupling requires more tests than simple software with high cohesion and loose coupling. Without unit tests, the costs of the poor design are borne by QA, operations, and customers. With unit tests, the costs are borne by the programmers. Unit tests require time and effort to write, and at their best programmers are lazy and proud folk.[1] They don't want to spend time writing needless tests.

1. Laziness is defined by Larry Wall as the quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it.

Unit tests make low cohesion visible through the costs of test setup. Low cohesion increases the number of setup tasks performed in a test. In a functionally cohesive module, it is usually only necessary to set up a few different sets of test conditions. The code to set up such a condition is called a *test fixture*. In a random or functionally cohesive module, many more fixtures are required by comparison. Each fixture is code that must be written, and time and effort that must be expended.

The more dependencies on external modules, the more setup is required for tests, and the more tests must be written. Each different class of inputs has to be tested, and each different class of input is yet another test to be written.

Methods with many inputs frequently have complicated logic, and each path through a method has to be tested. A single execution path mandates one test, and from there it gets worse. Each if-then statement increases the number of tests by two. Complicated loop bodies increase setup costs. The number of classes of output from a method also increases the number of tests to be performed as each kind of value returned and exception raised must be tested.

In a tightly coupled system, individual tests must reference many modules. The test writer expends effort setting up fixtures for each test. Over and over, the programmer confronts the external dependencies. The tests get ugly and the fixtures proliferate. The cost of tight coupling becomes apparent. A simple quantitative analysis shows the difference in testing effort between two designs.

Consider two methods named `get_urls()` that implement the same functionality. One has multiple return types, and the other always returns lists. In the first case, the method can return `None`, a single URL, or a nonempty array of URLs. We'll need at least three tests for this method—one for each distinct return value.

Now consider a method that consumes results from `get_urls()`. I'll call it `get_content(url_list)`. It must be tested with three separate inputs—one for each return type from `get_urls()`. To test this pair of methods, we'll have created six tests.

Contrast this with an implementation of `get_urls()` that returns only the empty array `[]` or a nonempty array of URLs. Testing `get_urls()` requires only two tests.

The associated definition for `get_content(url_list)` is correspondingly smaller, too. It just has to handle arrays, so it only requires one test, which brings the total to three. This is half the number of the first implementation, so it is immediately clear which interface is more complicated. What before seemed like a relatively innocuous choice now seems much less so.

Unit testing works with a programmer's natural proclivities toward laziness, impatience, and pride. It also improves design by facilitating refactoring.

Refactorings alter the structure of the code without altering its function. They are used to improve existing code. They are applied serially, and the unit tests are run after each one. If the behavior of the system has changed in unanticipated ways, then the test suite breaks. Without unit tests, the programmer must take it as an article of faith that the program's behavior is unchanged. This is foolish with your own code, and nearly insane with another's.

The Problems with Not Unit Testing

I make the bald-faced assertion that no programmer completely understands any system of nontrivial complexity. If that programmer existed, then he would produce completely bug-free code. I've yet to see that in practice, but absence of evidence is not evidence of

absence, so that person might exist. Instead, I think that programmers understand most of the salient features of their own code, and this is good enough in the real world.

What about working with another programmer's code? While you may understand the salient features of your code, you must often guess at the salient features of another's. Even when she documents her intent, things that were obvious to her may be perplexing to you. You don't have access to her thoughts. The design trade-offs are often opaque. The reasons for putting this method here or splitting out that method there may be historical or related to obscure performance issues. You just don't know for sure. Without unit tests or well-written comments, this can lead to pathological situations.

I've worked on a system where great edifices were constructed around old, baroque code because nobody dared change it. The original authors were gone, and nobody understood those sections of the code base. If the old code broke, then production could be taken down. There was no way to verify that refactorings left the old functionality unaltered, so those sections of code were left unchanged. Scope for projects was narrowly restricted to certain components, even if changes were best made in other components. Refactoring old code was strongly avoided.

It was the opposite of the ideal of collective code ownership, and it was driven by fear of breaking another's code. An executable test harness written by the authors would have verified when changes broke the application. With this facility, we could have updated the code with much less fear. Unit tests are a key to collective code ownership, and the key to confident and successful refactorings.

Code that isn't refactored constantly rots. It accumulates warts. It sprouts methods in inappropriate places. New methods duplicate functionality. The meanings of method and variable names drift, even though the names stay the same. At best, the inappropriate names are amusing, and at worst misleading.

Without refactoring, local bugs don't stay restricted to their neighborhoods. This stems from the layering of code. Code is written in layers. The layers are structural or temporal. Structural layering is reflected in the architecture of the system. Raw device IO calls are invoked from buffered IO calls. The buffered IO calls are built into streams, and applications sip from the streams. Temporal layering is reflected in the times at which features are created. The methods created today are dependent upon the methods that were written earlier. In either case, each layer is built upon the assumption that lower layers function correctly.

The new layers call upon previous layers in new and unusual ways, and these ways uncover existing but undiscovered bugs. These bugs must be fixed, but this frequently means that overlaying code must be modified in turn. This process can continue up through the layers as each in turn must be altered to accommodate the changes below them. The more tightly coupled the components are, the further and wider the changes will ripple through the system. It leads to the effect known as *collateral damage* (a.k.a. *whack-a-mole*), where fixing a bug in one place causes new bugs in another.

Pessimism

There are a variety of reasons that people condemn unit testing or excuse themselves from the practice. Some I've read of, but most I've encountered in the real world, and I recount those here.

One common complaint is that unit tests take too long to write. This implies that the project will take longer to produce if unit tests are written. But in reality, the time spent on unit

testing is recouped in savings from other places. Much less time is spent debugging, and much less time is spent in QA. Extensively unit-tested projects have fewer bugs. Consequently, less developer and QA time is spent on repairing broken features, and more time is spent producing new features.

Some developers say that writing tests is not their job. What is a developer's job then? It isn't simply to write code. A developer's job is to produce working and completely debugged code that can be maintained as cheaply as possible. If unit tests are the best means to achieve that goal, then writing unit tests is part of the developer's job.

More than once I've heard a developer say that they can't test the code because they don't know how it's supposed to behave. If you don't know how the code is supposed to behave, then how do you know what the next line should do? If you really don't know what the code is supposed to do, then now probably isn't the best time to be writing it. Time would be better spent understanding what the problem is, and if you're lucky, there may even be a solution that doesn't involve writing code.

Sometimes it is said that unit tests can't be used because the employer won't let unit tests be run against the live system. Those employers are smart. Unit tests are for the development environment. They are the programmer's tools. Functional tests can run against a live system, but they certainly shouldn't be running against a production system.

The cry of "But it compiles!" is sometimes heard. It's hard to believe that it's heard, but it is from time to time. Lots of bad code compiles. Infinite loops compile. Pointless assignments compile. Pretty much every interesting bug comes from code that compiles.

More often, the complaint is made that the tests take too long to run. This has some validity, and there are interesting solutions. Unit tests should be fast. Hundreds should run in a second. Some unit tests take longer, and these can be run less frequently. They can be deferred until check-in, but the official build must always run them.

If the tests still take too long, then it is worth spending development resources on making them go faster. This is an area ripe for improvement. Test runners are still in their infancy, and there is much low-hanging fruit that has yet to be picked.

"We tried and it didn't work" is the complaint with the most validity. There are many individual reasons that unit testing fails, but they all come down to one common cause. The practice fails unless the tests provide more perceived reliability than they cost in maintenance and creation combined. The costs can be measured in effort, frustration, time, or money. People won't maintain the tests if the tests are deemed unreliable, and they won't maintain the tests unless they see the benefits in improved reliability.

Why does unit testing fail? Sometimes people attempt to write comprehensive unit tests for existing code. Creating unit tests for existing code is hard. Existing code is often unsuited to testing. There are large methods with many execution paths. There are a plethora of arguments feeding into functions and a plethora of result classes coming out. As I mentioned when discussing design, these lead to larger numbers of tests, and those tests tend to be more complicated.

Existing code often provides few points where connections to other parts of the system can be severed, and severing these links is critical for reducing test complexity. Without such access points, the subject code must be instrumented in involved and Byzantine ways. Figuring out how to do this is a major part of harnessing existing code. It is often easier just to rewrite the code than to figure out a way to sever these dependencies or instrument the internals of a method.

Tests for existing code are written long after the code is written. The programmer is in a different state of mind, and it takes time and effort to get back to that mental state where the code was written. Details will have been forgotten and must be deduced or rediscovered. It's even worse when someone else wrote the code. The original state of mind is in another's head and completely inaccessible. The intent can only be imperfectly intuited.

There are tools that produce unit tests from finished code, but they have several problems. The tests they produce aren't necessarily simple. They are as opaque, or perhaps more opaque, than the methods being tested. As documentation, they leave something to be desired, as they're not written with the intent to inform the reader. Even worse, they will falsely ensure the validity of broken code. Consider this code fragment:

```
a = a + y
a = a + y
```

The statement is clearly duplicated. This code is probably wrong, but currently many generators will produce a unit test that validates it.

An effort focused on unit testing unmodified existing code is likely to fail. Unit testing's big benefits accrue when writing new code. Efforts are more likely to succeed when they focus on adding unit tests for sections of code as they change.

Sometimes failure extends from a limited suite of unit tests. A test suite may be limited in both extent and execution frequency. If so, bugs will slip through and the tests will lose much of their value. In this context, *extent* refers to coverage within a tested section. Testing coverage should be as complete as possible where unit tests are used. Tested areas with sparse coverage leak bugs, and this engenders distrust.

When fixing problems, all locations evidencing new bugs must be unit tested. Every mole that pops out of its hole must be whacked. Fixing the whack-a-mole problem is a major benefit that developers can see. If the mole holes aren't packed shut, the moles will pop out again, so each bug fix should include an associated unit test to prevent its regression in future modifications.

Failure to properly fix broken unit tests is at the root of many testing effort failures. Broken tests must be fixed, not disabled or gutted.[2] If the test is failing because the associated functionality has been removed, then gutting a unit test is acceptable; but gutting because you don't want to expend the effort to fix it robs tests of their effectiveness. There was clearly a bug, and it has been ignored. The bug will come back, and someone will have to track it down again. The lesson often taken home is that unit tests have failed to catch a bug.

Why do people gut unit tests? There are situations in which it can reasonably be done, but they are all tantamount to admitting failure and falling back to a position where the testing effort can regroup. In other cases, it is a social problem. Simply put, it is socially acceptable in the development organization to do this. The way to solve the problem is by bringing social pressures to bear.

Sometimes the testing effort fails because the test suite isn't run often enough, or it's not run automatically. Much of unit testing's utility comes through finding bugs immediately after they are introduced. The longer the time between a change and its effect, the harder it is to associate the two. If the tests are not run automatically, then they won't be run much of the

2. A test is gutted when its body is removed, leaving a stub that does nothing.

time, as people have a natural inclination not to spend effort on something that repeatedly produces nonresults or isn't seen to have immediate benefits.

Unit tests that run only on the developer's system or the build system lead toward failure. Developers must be able to run the tests at will on their own development boxes, and the build system must be able to run them in the official clean build environment. If developers can't run the unit tests on their local systems, then they will have difficulty writing the tests. If the build system can't run the tests, then the build system can't enforce development policies.

When used correctly, unit test failures should indicate that the code is broken. If unit test failures do not carry this meaning, then they will not be maintained. This meaning is enforced through build failures. The build must succeed only when all unit tests pass. If this cannot be counted on, then it is a severe strike against a successful unit-testing effort.

Test-Driven Development

As noted previously, a unit-testing effort will fail unless the tests provide more perceived reliability than the combined costs of maintenance and creation. There are two clear ways to ensure this. Perceived utility can be increased, or the costs of maintenance and creation can be decreased. The practices of TDD address both.

TDD is a style with unique characteristics. Perhaps most glaringly, tests are written before the tested code. The first time you encounter this, it takes a while to wrap your mind around it. "How can I do that?" was my first thought, but upon reflection, it is obvious that you always know what the next line of code is going to do. You can't write it until you know what it is going to do. The trick is to put that expectation into test code before writing the code that fulfills it.

TDD uses very small development cycles. Tests aren't written for entire functions. They are written incrementally as the functions are composed. If the chunks get too large, a test-driven developer can always back down to a smaller chunk.

The cycles have a distinct four-part rhythm. A test is written, and then it is executed to verify that it fails. A test that succeeds at this point tells you nothing about your new code. (Every day I encounter one that works when I don't expect it to.) After the test fails, the associated code is written, and then the test is run again. This time it should pass. If it passes, then the process begins anew.

The tests themselves determine what you write. You only write enough code to pass the test, and the code you write should always be the simplest possible thing that makes the test succeed. Frequently this will be a constant. When you do this religiously, little superfluous functionality results.

No code is allowed to go into production unless it has associated tests. This rule isn't as onerous as it sounds. If you follow the previously listed practices then this happens naturally.

The tests are run automatically. In the developer's environment, the tests you run may be limited to those that execute with lightning speed (i.e., most tests). When you perform a full build, all tests are executed. This happens in both the developer's environment and the official build environment. A full build is not considered successful unless all unit tests succeed.

The official build runs automatically when new code is available. You've already seen how this is done with Buildbot, and I'll expand the configuration developed in Chapter 5 to include running tests. The force of public humiliation is often harnessed to ensure compliance. Failed builds are widely reported, and the results are highly visible. You often accomplish this through mailing lists, or a visible device such as a warning light or lava lamp.

Local test execution can also be automated. This is done through two possible mechanisms. A custom process that watches the source tree is one such option, and another uses the IDE itself, configuring it to run tests when the project changes.

The code is constantly refactored. When simple implementations aren't sufficient, you replace them. As you create additional functionality, you slot it into dummied implementations. Whenever you encounter duplicate functionality, you remove it. Whenever you encounter code smells, the offending stink is freshened.

These practices interact to eliminate many of the problems encountered with unit testing. They speed up unit testing and improve the tests' accuracy. The tests for the code are written at the same time the code is written. There are no personnel or temporal gaps between the code and the tests. The tests' coverage is exhaustive, as no code is produced without an associated set of tests. The tests don't go stale, as they are invoked automatically, and the build fails if any tests fail. The automatic builds ensure that bugs are found very soon after they are introduced, vastly improving the suite's value.

The tests are delivered with the finished system. They provide documentation of the system's components. Unlike written documents, the tests are verifiable, they're accurate, and they don't fall out of sync with the code. Since the tests are the primary documentation source, as much effort is placed into their construction as is placed into the primary application.

Knowing Your Unit Tests

A unit test must assert success or failure. Python provides a ready-made command. The Python `assert` expression takes one argument: a Boolean expression. It raises an `AssertionErrror` if the expression is `False`. If it is `True`, then the execution continues on. The following code shows a simple assertion:

```
>>> a = 2
>>> assert a == 2
>>> assert a == 3
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

You clarify the test by creating a more specialized assertion:

```
>>> def assertEquals(x, y):
... assert x == y
...
>>> a = 2
>>> assertEquals(a, 2)
>>> assertEquals(a, 3)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in assertEquals
AssertionError
```

Unit tests follow a very formulaic structure. The test conditions are prepared, and any needed fixtures are created. The subject call is performed, the behavior is verified, and finally the test fixtures are cleanly destroyed. A test might look like this:

```
def testSettingEmployeeNameShouldWork():
     x = create_persistent_employee()
     x.set_name("bob")
     assertEquals("bob", x.get_name)
     x.destroy_self()
```

The next question is where the unit tests should go. There are two reasonable choices: the tests can be placed with the code they test or in an isolated package. I personally prefer the former, but the latter has performance advantages and organizational benefits. The tools to run unit tests often search directories for test packages. For large projects, this overhead causes delays, and I'd rather sidestep the issue to begin with.

unittest and Nose

There are several packages for unit testing with Python. They all support the four-part test structure described previously, and they all provide a standard set of features. They all group tests, run tests, and report test results. Surprisingly, test running is the most distinctive feature among the Python unit-testing frameworks.

There are two clear winners in the Python unit-testing world: unittest and Nose. unittest ships with Python, and Nose is a third-party package. Pydev provides support for unittest, but not for Nose. Nose, on the other hand, is a far better test runner than unittest, and it understands how to run the other's test cases.

Like Java's jUnit test framework, unittest is based upon Smalltalk's xUnit. Detailed information on its development and design can be found in Kent Beck's book *Test-Driven Development: By Example* (Addison-Wesley, 2002).

Tests are grouped into `TestCase` classes, modules (files), and `TestSuite` classes. The tests are methods within these classes, and the method names identify them as tests. If a method name begins with the string `test`, then it is a test—so `testy`, `testicular`, and `testosterone` are all valid test methods. Test fixtures are set up and torn down at the level of `TestCase` classes. `TestCase` classes can be aggregated with `TestSuite` classes, and the resulting suites can be further aggregated. Both `TestCase` and `TestSuite` classes are instantiated and executed by `TestRunner` objects. Implicit in all of this are modules, which are the Python files containing the tests. I never create `TestSuite` classes, and instead rely on the implicit grouping within a file.

Pydev knows how to execute unittest test objects, and any Python file can be treated as a unit test. Test discovery and execution are unittest's big failings. It is possible to build up a giant unit test suite, tying together `TestSuite` after `TestSuite`, but this is time-consuming. An easier approach depends upon file-naming conventions and directory crawling. Despite these deficiencies, I'll be using unittest for the first few examples. It's very widely used, and familiarity with its architecture will carry over to other languages.[3]

3. Notably, it carries over to JavaScript testing with JSUnit in Chapter 10.

Nose is based on an earlier package named PyTest. Nose bills itself primarily as a test discovery and execution framework. It searches directory trees for modules that look like tests. It determines what is and is not a test module by applying a regular expression (`r'(?:^|[\\b_\\.%s-])[Tt]est' % os.sep`) to the file name. If the string `[Tt]est` is found after a word boundary, then the file is treated as a test.[4] Nose recognizes `unittest.TestCase` classes, and knows how to run and interpret their results. `TestCase` classes are identified by type rather than by a naming convention.

Nose's native tests are functions within modules, and they are identified by name using the same pattern used to recognize files. Nose provides fixture setup and tear-down at both the module level and function level. It has a plug-in architecture, and many features of the core package are implemented as plug-ins.

A Simple RSS Reader

The project introduced in Chapter 4 is a simple command-line RSS reader (a.k.a. aggregator). As noted, RSS is a way of distributing content that is frequently updated. Examples include new articles, blog postings, podcasts, build results, and comic strips. A single source is referred to as a feed. An aggregator is a program that pulls down one or more RSS feeds and interleaves them. The one constructed here will be very simple. The two feeds we'll be using are from two of my favorite comic strips: xkcd and PVPonline.

RSS feeds are XML documents. There are actually three closely related standards: RSS, RSS 2.0, and Atom. They're more alike than different, but they're all slightly incompatible. In all three cases, the feeds are composed of dated items. Each item designates a chunk of content. Feed locations are specified with URLs, and the documents are typically retrieved over HTTP.

You could write software to retrieve an RSS feed and parse it, but others have already done that work. The well-recognized package FeedParser is one. It is retrieved with `easy_install`:

```
$ easy_install FeedParser
```

```
Searching for FeedParser
Reading http://pypi.python.org/simple/FeedParser/
Best match: feedparser 4.1
...
Processing dependencies for FeedParser
Finished processing dependencies for FeedParser
```

The package parses RSS feeds through several means. They can be retrieved and read remotely through a URL, and they can be read from an open Python file object, a local file name, or a raw XML document that can be passed in as a string. The parsed feed appears as a queryable data structure with a `dict`-like interface:

4. The default test pattern recognizes `Test.py`, `Testerosa.py`, `a_test.py`, and `testosterone.py`, but not `CamelCaseTest.py` or `mistested.py`. You can set the pattern with the `-m` option.

```
>>> import feedparser
>>> d = feedparser.parse('http://www.xkcd.com/rss.xml')
>>> print d['feed']['title']
```

```
xkcd.com
```

```
>>> print len(d['items'])
```

```
2
```

```
>>> print [x['title'] for x in d['items']]
```

```
[u'Python', u'Far Away']
```

```
>>> print [x['date'] for x in d['items']]
```

```
[u'Wed, 05 Dec 2007 05:00:00 -0000', u'Mon, 03 Dec 2007➥
05:00:00 -0000']
```

The project is ill defined at this point, so I'm going to describe it a bit more concretely. We'll start simply and add more features as the project develops. For now, I just want to know if a new comic strip is available when I log in. (I find it really depressing to get the Asia Times feed in the morning, and comics make me happy.)

Let's make a story. User stories describe new features. They take the place of large requirements documents. They are only two or three sentences long and have just enough detail for a developer to make a ballpark estimate of how long it will take to implement. They're initially created by the customer, they're devoid of technical mumbo jumbo, and they're typically jotted down on a note card, as in Figure 6-1.

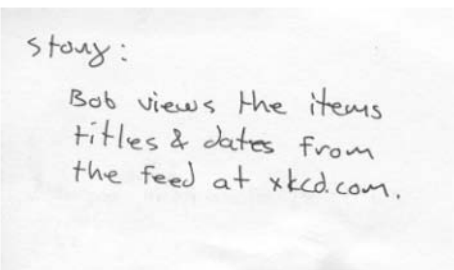


Figure 6-1. *A user story on a 3 × 5 notecard*

Developers go back to the customer when work begins on the story. Further details are hashed out between the two of them, ensuring that the developer really understands what the customer wants, with no intermediate document separating their perceptions. This discussion's outcomes drive acceptance test creation. The acceptance tests document the discussion's conclusions in a verifiable way.

In this case, I'm both the customer and the programmer. After a lengthy discussion with myself, I decide that I want to run the command with a single URL or a file name and have it output a list of articles. The user story shown on the card in Figure 6-1 reads, "Bob views the titles & dates from the feed at xkcd.com." After hashing things out with the customer, it turns out that he expects a run to look something like this:

```
$ rsreader http://www.xkcd.com/rss.xml
```

```
Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away
```

I ask the customer (me), "What should this look like when I don't supply any arguments?" And the customer says, "Well, I expect it to do nothing."

And the developer (me) asks, "And if it encounters errors?"

"Well, I really don't care about that. I'm a Python programmer. I'll deal with the exceptions," replies the customer, "and for that matter, I don't care if I even see the errors."

"OK, what if more than one URL is supplied?"

"You can just ignore that for the moment."

"Cool. Sounds like I've got enough to go on," and remembering that maintaining good relations with the customer is important, I ask, "How about grabbing a bite for lunch at China Garlic?"

"Great idea," the customer replies.

We now have material for a few acceptance tests. The morning's work is done, and I go to lunch with myself and we both have a beer.

The First Tests

In the previous chapter, you wrote a tiny fragment of code for your application. It's a stub method that prints "woof." It exists solely to allow Setuptools to install an application. The project (as seen from Eclipse) is shown in Figure 6-2.

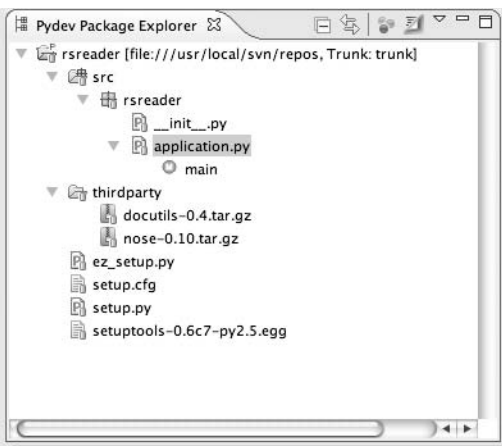


Figure 6-2. *RSReader as last visited*

Instead of intermixing test code and application code, the test code is placed into a separate package hierarchy. The package is `test`, and there is also a test module called `test.test_application.py`. This can be done from the command line or from Eclipse. The added files and directories are shown in Figure 6-3.

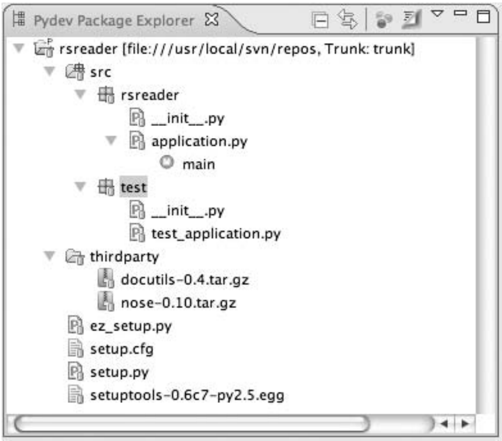


Figure 6-3. *RSReader with the unit test skeleton added*

RSReader takes in data from URLs or files. The acceptance tests shouldn't depend on external resources, so the first acceptance tests should read from a file. They will expect a specific output, and this output will be hard-coded. The method `rsreader.application.main()` is the application entry point defined in `setup.py`. You need to see what a failing test looks like before you can appreciate a successful one, so the first test case initially calls `self.fail()`:

```
from unittest import TestCase

class AcceptanceTests(TestCase):

    def test_should_get_one_URL_and_print_output(self):
        self.fail()
```

The test is run through the Eclipse menus. The test module is selected from the Package Explorer pane, or the appropriate editor is selected. With the focus on the module, the Run menu is selected from either the application menu or the context menu. From the application menu, the option is Run ➤ Run As ➤ "Python unit-test," and from the context menu, it is Run As ➤ "Python unit-test." Once run, the console window will report the following:

```
Finding files... ['/Users/jeff/workspace/rsreader/src/test/test_application.py']➥
 ... done
Importing test modules ... done.

test_should_get_one_URL_and_print_output➥
 (test_application.AcceptanceTests) ... FAIL
```

```
======================================================================
FAIL: testShouldGetOneURLAndPrintOutput (test_application.AcceptanceTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/jeff/workspace/rsreader/src/test/test_application.py",➥
 line 6, in testShouldGetOneURLAndPrintOutput
    self.fail()
AssertionError

----------------------------------------------------------------------
Ran 1 test in 0.000s
```

The output shows that one test was run, and it took less than 1 ms. As expected, the test failed.

This example starts with a bang; a very complicated bang. Feeding input into the program and reading the output is the most complicated thing done in this chapter. The print statement writes to `sys.stdout`. The test should capture `sys.stdout`, and then compare the output with the expectations.

`sys.stdout` contains a file-like object. The test replaces this object with a `StringIO` instance. `StringIO` is a file-like object that accumulates written information in a string. This string's value can be extracted and compared with the expected value.

Care must be taken when doing this. If the old value of `sys.stdout` is not restored, then it will be lost, and no more output will be reported. Instead of going to the console, the output will accumulate in the inaccessible `StringIO` object. A first pass looks something like this:

```
import StringIO
import sys
from unittest import TestCase

from rsreader.application import main

class AcceptanceTests(TestCase):

    def test_should_get_one_URL_and_print_output(self):
        printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""

        old_value_of_stdout = sys.stdout
        try:
            sys.stdout = StringIO.StringIO()
            main()
            self.assertEquals(printed_items + "\n",
                              sys.stdout.getvalue())
        finally:
            sys.stdout = old_value_of_stdout
```

The core statements of the test are in bold. When run, this test fails as expected. The important line of output reads as follows:

```
AssertionError: 'Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com:➥
 Python\nMon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away\n' !=➥
 'woof\n'
```

As hoped, the `printed_items` list does not match the recorded output. The test shows that the output, `woof`, was indeed captured, though. The most questionable part of the test mechanics has been checked.

The test isn't complete, though. The URL needs to be passed in through `sys.argv`. `sys.argv` is a list, and the first argument of the list is always the name of the program—that's just how it works. The single URL will be the second element in the list. `sys.argv` is also a global variable, so it needs the same treatment as `sys.stdout`:

```
class AcceptanceTests(TestCase):

    def test_should_get_one_URL_and_print_output(self):
        printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""

        old_value_of_stdout = sys.stdout
        old_value_of_argv = sys.argv
        try:
            sys.stdout = StringIO.StringIO()
            sys.argv = ["unused_prog_name", "xkcd.rss.xml"]
            main()
            self.assertEquals(printed_items + "\n",
                              sys.stdout.getvalue())
        finally:
            sys.stdout = old_value_of_stdout
            sys.argv = old_value_of_argv
```

Running the method shows the same results as before—the test fails with an equality mismatch.

The test method is complete. Now, what is the simplest possible way to make the code pass the test? The simplest way is by faking the results. The new code is shown in bold.

```
def main():
    xkcd_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    print xkcd_items
```

This change is saved, and the test case is run again:

```
Finding files... ['/Users/jeff/Documents/ws/rsreader/src/test/➥
test_application.py'] ... done
Importing test modules ... done.

test_should_get_one_URL_and_print_output➥
 (test_application.AcceptanceTests) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

The test case has become pretty gruesome. Fixtures are set up, and if the setup succeeds, then they must be torn down afterward. This is part of the standard pattern described earlier, though, and unittest addresses these situations. It provides a mechanism to remove this code from the test case. This uses the magical `setUp(self)` and `tearDown(self)` methods. If defined, they are called at the beginning and end of every unit test. `TearDown()` will only be skipped under one condition, and that is when `setUp()` is defined yet fails. In that case, the entire test is abandoned.

I'll demonstrate the refactoring in two steps. First, the `sys.stdout` code will be moved into the `setUp()` and `tearDown()` methods, and then the `sys.argv` code will be moved to them:

```
class AcceptanceTests(TestCase):

    def setUp(self):
        self.old_value_of_stdout = sys.stdout
        sys.stdout = StringIO.StringIO()

    def tearDown(self):
        sys.stdout = self.old_value_of_stdout

    def test_should_get_one_URL_and_print_output(self):
        printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""

        old_value_of_argv = sys.argv
        try:
            sys.argv = ["unused_prog_name", "xkcd.rss.xml"]
            main()
            self.assertEquals(printed_items + "\n",
                              sys.stdout.getvalue())
        finally:
            sys.argv = old_value_of_argv
```

Running this test succeeds. With the assurance that nothing is broken, the second refactoring is performed:

```
class AcceptanceTests(TestCase):

    def setUp(self):
        self.old_value_of_stdout = sys.stdout
        sys.stdout = StringIO.StringIO()
        self.old_value_of_argv = sys.argv

    def tearDown(self):
        sys.stdout = self.old_value_of_stdout
        sys.argv = self.old_value_of_argv

    def test_should_get_one_URL_and_print_output(self):
        printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""

        sys.argv = ["unused_prog_name", "xkcd.rss.xml"]
        main()
        self.assertEquals(printed_items + "\n", sys.stdout.getvalue())
```

Running the test again demonstrates that nothing has changed. The test still passes, and the test is notably cleaner. The `try` block has been removed, and the test method retains only code related to the test itself.

The next test focuses on empty input. Casting back to the use case discussion, there should be no output when there are no URLs or files specified. The test for that condition is quite compact:

```
    def test_no_urls_should_print_nothing(self):
        sys.argv = ["unused_prog_name"]
        main()
        self.assertEquals("", sys.stdout.getvalue())
```

Running the test produces the following output:

```
Importing test modules ... done.

test_no_urls_should_print_nothing➥
(test_application.AcceptanceTests) ... FAIL
test_should_get_one_URL_and_print_output➥
 (test_application.AcceptanceTests) ... ok

======================================================================
FAIL: test_no_urls_should_print_nothing (test_application.AcceptanceTests)
----------------------------------------------------------------------
Traceback (most recent call last):
```

```
  File "/Users/jeff/Documents/ws/rsreader/src/test/test_application.py",➥
line 30, in test_no_urls_should_print_nothing
    self.assertEquals("\n", sys.stdout.getvalue())
AssertionError: '\n' != 'Wed, 05 Dec 2007 05:00:00 -0000:➥
xkcd.com: Python\nMon, 03 Dec 2007 05:00:00 -0000: xkcd.com:➥
 Far Away\n'

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

Summaries for both tests are shown. It is clear that the new test failed, while the old test succeeded. The new test failed because the hard-coded output is a constant. The main routine needs to be changed. It should print nothing when no user arguments are passed. `main()` only needs to distinguish between two options, so we fake it a little more:

```
def main():
    xkcd_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    if len(sys.argv) == 2:
        print xkcd_items
```

The tests are run, and the second test now passes. There is a third acceptance test that was discussed. In that case, more than two feeds are passed in, but only the first is reported. The new test case reads as follows:

```
    def test_many_urls_should_print_first_results(self):
        printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""

        sys.argv = ["unused_prog_name", "xkcd.rss.xml", "excess"]
        main()
        self.assertEquals(printed_items + "\n", sys.stdout.getvalue())
```

The test is run, and it fails. It fails because the `main()` method explicitly checks for a length of 2. In all other cases, it prints nothing. This is corrected by checking for any nonempty user argument list:

```
def main():
    xkcd_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    if sys.argv[1:]:
        print xkcd_items
```

With this change, all three acceptance tests pass. There is now a solid framework for writing the rest of the application. At this point, the application can be installed with `python ./setup.py install`, or the local installation can be put into development mode with `python ./setup.py develop`, and the application runs. This can be verified from the command line. It's running in a brain-dead, bogus, dirt-simple way, but it can be refined into useful functionality from there.

```
$ rsreader
$ rsreader xkcd.rss
```

```
Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away
```

```
$ rsreader xkcd.rss something.useless
```

```
Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away
```

There are still things to clean up. Tests will have to be rewritten in the future, so they must be maintainable. Tests serve as documentation, too, so they must also be readable. They obey the same rules as the application code, and if refactoring is neglected, then the tests will rot.

There are two duplications within the tests. The constant `printed_items` can be lifted out of the first and third tests, and the lines comparing the captured `sys.stdout` can be extracted into a new method, which I'll call `assertStdoutEquals`. After these refactorings, the tests look like this:

```
class AcceptanceTests(TestCase):

    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""

    def setUp(self):
        self.old_value_of_stdout = sys.stdout
        sys.stdout = StringIO.StringIO()
        self.old_value_of_argv = sys.argv

    def tearDown(self):
        sys.stdout = self.old_value_of_stdout
        sys.argv = self.old_value_of_argv

    def test_should_get_one_URL_and_print_output(self):
        sys.argv = ["unused_prog_name", "xkcd.rss.xml"]
        main()
        self.assertStdoutEquals(self.printed_items + "\n")
```

```
    def test_no_urls_should_print_nothing(self):
        sys.argv = ["unused_prog_name"]
        main()
        self.assertStdoutEquals("")

    def test_many_urls_should_print_first_results(self):
        sys.argv = ["unused_prog_name", "xkcd.rss.xml", "excess"]
        main()
        self.assertStdoutEquals(self.printed_items + "\n")

    def assertStdoutEquals(self, expected_output):
        self.assertEquals(expected_output, sys.stdout.getvalue())
```

These tests are run, they still execute, and the application still runs. The smell of duplication has been removed, but there are still things to be done before work can begin on the rest of the application.

Finding Tests with Nose

Running tests manually within Eclipse is fine for a brief tutorial, but it doesn't cut it for day-to-day work. Triggering the tests takes time and attention, and it breaks flow. This is a no-no. Tests should run automatically in the local development environment, and they must run automatically in the official build environment.

Running the unit tests cannot be automated unless it can be done from the command line. unittest does this very poorly, but that's all right. As noted earlier in this chapter, the Nose test package is much better at doing this.

The Nose test runner is `nosetests`. By default, it searches for tests starting in the current working directory. It identifies tests by name, and it only searches for them in valid Python packages. The search begins in a directory, and it extends recursively into any subdirectories that contain `__init__.py` files. Nose runs unittest `TestCase` classes, so we can use it to run the current acceptance tests from the command line. The next few lines are executed from the project root:

```
$ nosetests
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.048s

OK
```

Other directories are specified using the `-w` switch. This is particularly useful when working on a large project where searching for packages consumes a noticeable amount of time, or when for one reason or another, Nose is being run from a working directory that is not the project root.

```
$ nosetests -w src/test
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.048s

OK
```

By default, `nosetests` is quiet. It prints one dot for each successful test. An `F` is printed if a test assertion fails, and an `E` is printed if a test has an error that prevents it from running to completion. Stack traces are printed when a test fails or errors out. Developers quickly acquire an addiction to watching the little dots stream across the page.

Nose is made more vociferous using the `-v` switch. Instead of printing a string of dots, it prints one line for each test. It prints the test name and test module, or the doc string followed by a status that may be one of `ok`, `fail`, or `error`.

```
$ nosetests -w src/test -v
```

```
test_many_urls_should_print_first_results (test.test_application.➥
AcceptanceTests) ... ok
test_no_urls_should_print_nothing (test.test_application.➥
AcceptanceTests) ... ok
test_should_get_one_URL_and_print_output (test.test_application.➥
AcceptanceTests) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.002s
```

Nose also intercepts `stdout` and `stderr`, which sometimes isn't desired. Sometimes you'll need to see messages propagated by other modules, and other times you'll want to watch debugging messages you've inserted. At times like these, you can turn off output capture with the `-s` switch.

Skipping Slow Tests

The majority of tests in a test suite will run in a matter of seconds, but a small minority will take seconds or tens of seconds. This is far too long for the development environment. As long as the continuous build system takes up the slack, and as long as code is committed regularly, it is usually a win to skip these slow tests for most runs.

Nose provides the facility to do this through attribute tags. *Attribute Tags* are bits of metadata attached to test methods. In actuality, they're just additional attributes. I'll add the following test to demonstrate the feature:

```
def test_simulates_performing_a_timeout(self):
    import time
    time.sleep(5)
```

Running `nosetests` shows that the test runs, and it is clear that it significantly lengthens the suite's runtime:

```
$ nosetests -w src/test
```

```
....
----------------------------------------------------------------------
Ran 4 tests in 5.004s

OK
```

The next code snippet shows the slow test with a tag attached:

```
def test_simulates_performing_a_timeout(self):
    import time
    time.sleep(5)
test_simulates_performing_a_timeout.slow = True
```

`nosetests`'s `-a` option gives the tags meaning. It runs all tests matching an expression:

```
$ nosetests -w src/test -a slow
```

```
.
----------------------------------------------------------------------
Ran 1 test in 5.003s

OK
```

This is exactly the opposite of what you wanted. The option can be negated by prefixing it with an exclamation point. Most shells attach meaning to `!`, so it must be escaped with a backslash:

```
$ nosetests -w src/test -a \!slow
```

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
```

This is the desired result. Using attribute tags, the slow tests are designated, and they are skipped with the `-a` option, but the specification is pretty ugly. Fortunately, Python's decorators come to the rescue.

Decorators were introduced in Python 2.4. They are functions that (in the simplest case) take a function as input and return a function as output. They are used to modify existing methods. The most familiar usage specifies class methods. The following code defines a decorator that sets the `slow` attribute to `True`:

```
def slow(f):
    f.slow = True
    return f
```

Using this decorator, the tagged test becomes

```
@slow
def test_simulates_performing_a_timeout(self):
    import time
    time.sleep(5)
```

I find this definition much clearer, and running the previous Nose command proves that it works.

The `-a` option is implemented through a standard Nose plug-in named `attrib`. It has more features than I've demonstrated so far. In addition to testing a tag's existence, the tag's value may be checked for equality or inequality. The tag's value may also be a sequence. In this case, the tag expression checks for membership. This behavior is summarized in Table 6-1.

Table 6-1. *Tag Expressions*

Tag Option	Tag on Class	Executed by Tests?
`-a slow`	`test.slow = True`	Yes
`-a slow`	`test.slow = False`	No
`-a slow`	`test`	No
`-a \!slow`	`test.slow = True`	No
`-a \!slow`	`test.slow = False`	Yes
`-a \!slow`	`test`	Yes
`-a slow=a`	`test.slow="a"`	Yes
`-a slow=a`	`test.slow="b"`	No
`-a slow=a`	`test`	No
`-a slow=a`	`test.slow = ["a", "b"]`	Yes
`-a slow=a`	`test.slow = ["b"]`	No
`-a slow=\!a`	`test.slow = ["a"]`	No
`-a slow=\!a`	`test.slow="a"`	No
`-a slow\!=a`	`test.slow="a"`	No
`-a slow=\!a`	`test`	Yes

Integrating the Tests into the Environment

There are three times unit tests need to be run. First, they need to be run when changes are saved in the development environment. The test run should be fast, so this set of tests is restricted to those that run in a matter of seconds. Sometimes this means restricting them to a local area of the project, but often it means simply skipping those that take too long to run.

The full unit test suite should be run before changes are committed. This ensures that the code submitted works completely in the developer's environment. It catches bugs that show up in slow tests, and prevents them from reaching the shared codeline. It gives developers confidence that their changes won't break the build.

Finally, the official build system must run the complete unit test suite as part of every build, and the build must fail if any unit tests fail. This ensures that all tests pass in a clean environment. This also provides a mechanism to police unit-testing policy. Purely social pressures help, but the humiliation of a broken build is the big stick.

Running Tests After Every Change

Eclipse contains a mechanism intended to produce incremental builds. Eclipse activates builders when projects change. *Builders* take a list of changes since their last invocation, and then perform an update task. This may be an extension to Eclipse or a program run external to the IDE. The builder mechanism is a hook to run the unit tests after every change.

Additional builders are defined from the project properties menu. The project properties are accessed through the application menu or the context menu when the project is selected in the Package Explorer. From the application menu, the menu item is Project ➤ Properties. From the context menu, the menu item is Properties. This opens the window shown in Figure 6-4.

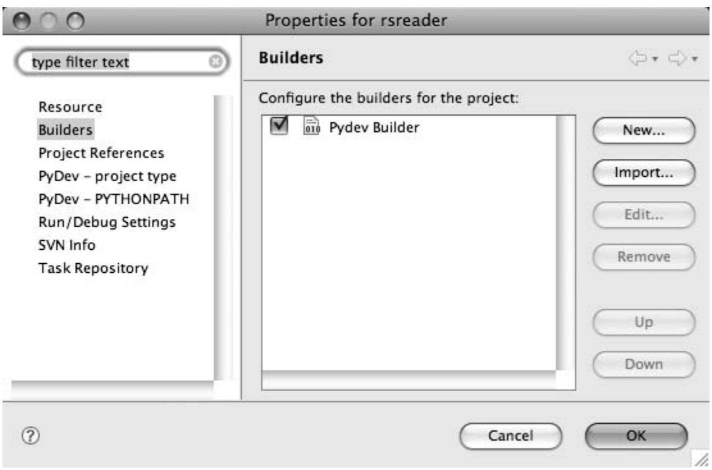


Figure 6-4. *The project properties window*

The Builders menu item is selected from the menu on the left, which brings up the panel shown. Clicking the New button brings up the window shown in Figure 6-5, from which the kind of builder is chosen.

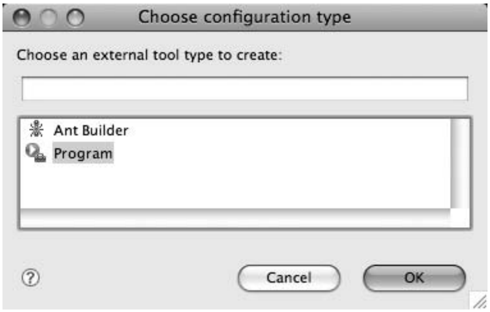


Figure 6-5. *Choosing the kind of external builder to create*

Out of the box, Eclipse offers two choices: Ant Builder and Program. Ant Builder calls Java's Ant build tool, and it understands how to parse the output, giving much more interesting output. I personally work on one mostly Python project that uses Ant as a build harness for historical reasons, but I prefer to use Setuptools when given the chance.

Ant is not being used in this project, though. Instead, tests are run through Nose, so the generic Program option is the correct choice. Clicking the OK button brings up the builder properties window, shown in Figure 6-6.

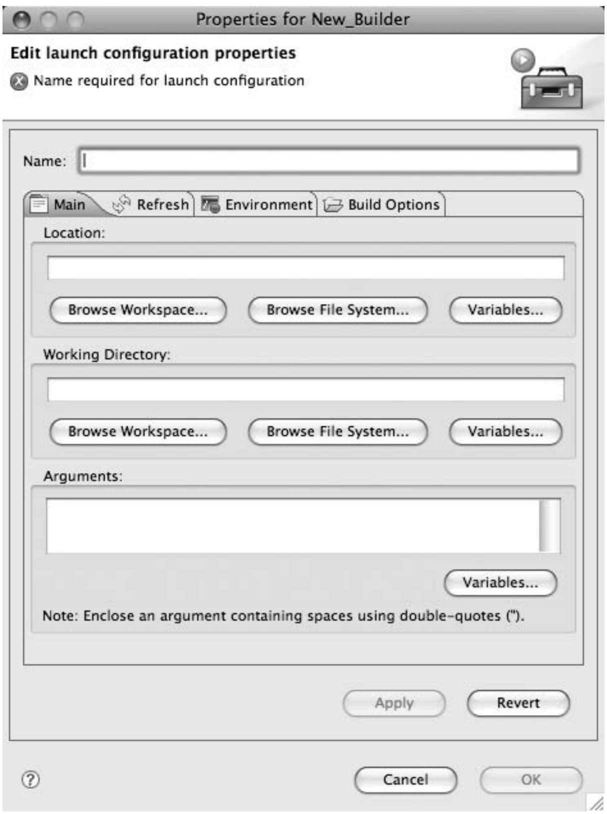


Figure 6-6. *The builder properties window*

The builder requires a name, so I'll call it Unit Tests. This name is for human consumption, and it has no significance to the IDE. In the Main tab, there are three fields to be filled in:

Location: This is the path to the `nosetests` binary. On UNIX systems, this can be found by executing the command `which nosetests`. On my Mac OS X system, the path is `/Users/jeff/bin/nosetests`, as specified by my `~/.pydistutils.cfg` file.

Working Directory: This is the top-level project directory. This is specified using the Eclipse variable expression `${workspace_loc:/rsreader}`. Using the Browse Workspace button to select this directory gives the same results.

Arguments: In this field, four options are passed: `-w src/test -v -s -a \!slow`. The option `-w src/test` specifies that Nose should only look for tests in the test directory. The option `-v` yields verbose output, showing all the tests, and the `-s` option ensures that any interesting output is sent to the console. The `-a \!slow` option specifies that only fast tests will be run (i.e., tests not marked as `slow`).

By default, the builder is only invoked after a clean build and during manual builds. For this purpose, it should be run when autobuilds are triggered, which happens after any change to the project. This setting is changed on the Build Options tab, which is shown in Figure 6-7.

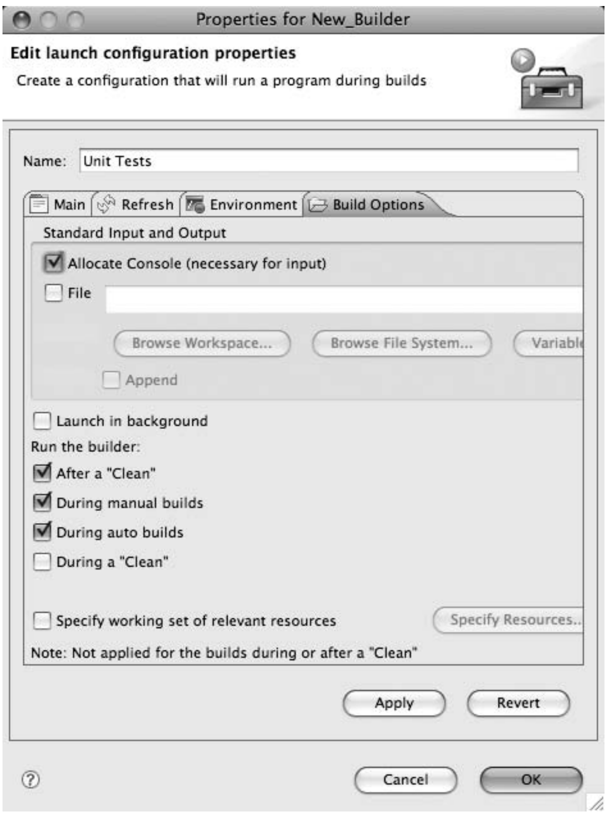


Figure 6-7. *The Build Options tab*

In Figure 6-7, the "During auto builds" check box has already been selected. It is the last selected check box in the window. If the unit tests take too long to execute, or you find that they interrupt your flow too much, then they may be backgrounded by default. This option is just below the console settings at the top. A console should always be allocated so that the results are clearly visible, and this is also the default.

With this change, the builder definition is complete. Clicking the OK button closes the builder properties window, and the focus returns to the project properties window shown in Figure 6-8.

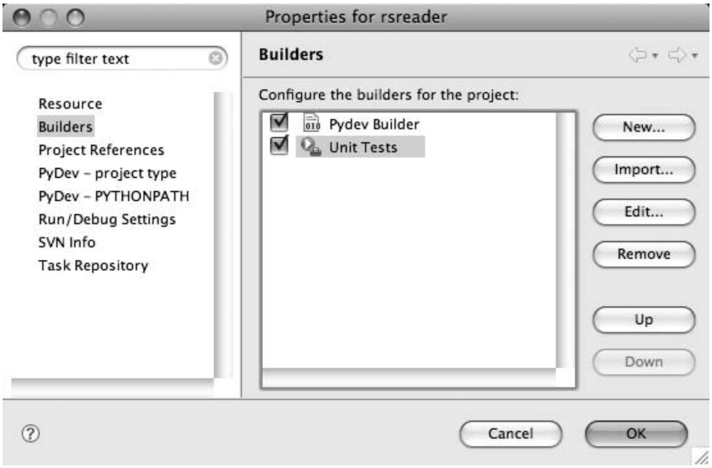


Figure 6-8. *The Unit Tests builder is defined and active.*

The Unit Tests builder is now available in the Builders list, and the check box to its left indicates that it is active. The order in the window determines when each builder is invoked, with the first invoked at the top and the last invoked at the bottom. You can reorder the list by selecting a builder and using the Up and Down buttons to change its position in the list.

Clicking OK saves the changes. This constitutes a change in the project, so the new builder launches immediately, and the test output is shown in the console, as in Figure 6-9.

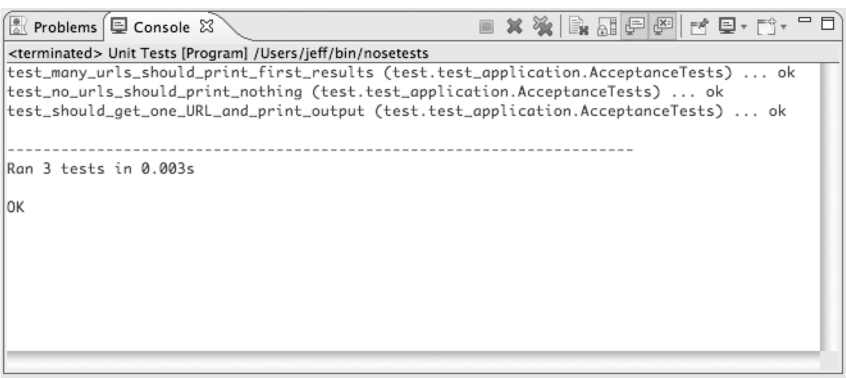


Figure 6-9. *The console showing the Unit Tests builder output*

Running the Complete Test Suite in Development

The complete test suite should be run before changes are committed. This is done through the build harness. This complete test suite check can be triggered from the command line or through Eclipse on demand. The changes to the harness are minimal and restricted to one line in `setup.py`.

Setuptools supports running unit tests through the command `setup.py test`. This command builds the package locally, and then runs a specified set of tests against this local installation. The tests are specified through the `test_suite` property. Nose provides a test collector that plugs into this property. The change to `setup.py` is shown here:

```
...
install_requires = [
                    'docutils == 0.4',
                    'nose == 0.10.0',
                    ],

# use nose to run tests
test_suite='nose.collector',

# metadata for upload to PyPI
...
```

Notice that `setup.py` already requires Nose, so it is always guaranteed to be there. The dependency could have been specified through the `tests_require` property. This property specifies packages that will only be installed for testing. Had Nose been installed through those routes, it would not be generally available for development work. Nose isn't required for production, but it doesn't hurt to bundle it along, and it makes it much easier to set up a development environment.

```
$ python ./setup.py test
```

```
running test
running egg_info
writing requirements to src/RSReader.egg-info/requires.txt
writing src/RSReader.egg-info/PKG-INFO
writing top-level names to src/RSReader.egg-info/top_level.txt
writing dependency_links to src/RSReader.egg-info/dependency_links.txt
writing entry points to src/RSReader.egg-info/entry_points.txt
writing manifest file 'src/RSReader.egg-info/SOURCES.txt'
running build_ext
test_many_urls_should_print_first_results (test.test_application.➥
AcceptanceTests) ... ok
test_no_urls_should_print_nothing (test.test_application.➥
AcceptanceTests) ... ok
test_should_get_one_URL_and_print_output (test.test_application.➥
AcceptanceTests) ... ok
```

```
----------------------------------------------------------------------
Ran 3 tests in 0.014s

OK
```

The next step makes the full test run available through Eclipse. Eclipse can run arbitrary programs from within the IDE and report the results. Eclipse calls these programs *external tools*. External tools are created and run through the application menu or the external tools button and drop-down on the toolbar. The external tools option is the little green play button with a toolbox in the lower-right-hand corner. It is shown in Figure 6-10.

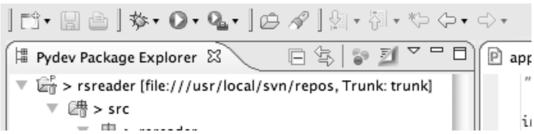


Figure 6-10. *The external tools button on the toolbar*

You create a new external tool through the application menu by selecting the Run ➤ External Tools ➤ External Tools Dialog menu item. From the drop-down button, the menu item is External Tools Dialog. This brings up the dialog shown in Figure 6-11.

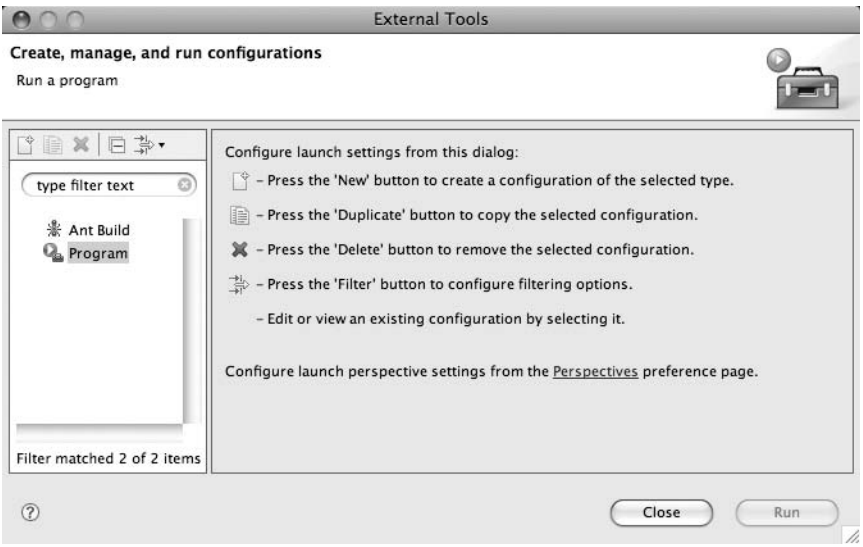


Figure 6-11. *The External Tools dialog*

The right half of the window contains basic instructions for getting started. The symbols there refer to the toolbar on the left. As with builders, there are two categories. One invokes Java's Ant and interprets the results, and the other executes arbitrary programs such as Python.

Clicking the leftmost icon on the toolbar or double-clicking the Program menu item creates a new program configuration and replaces the instructions with an editing panel. This is shown in Figure 6-12.

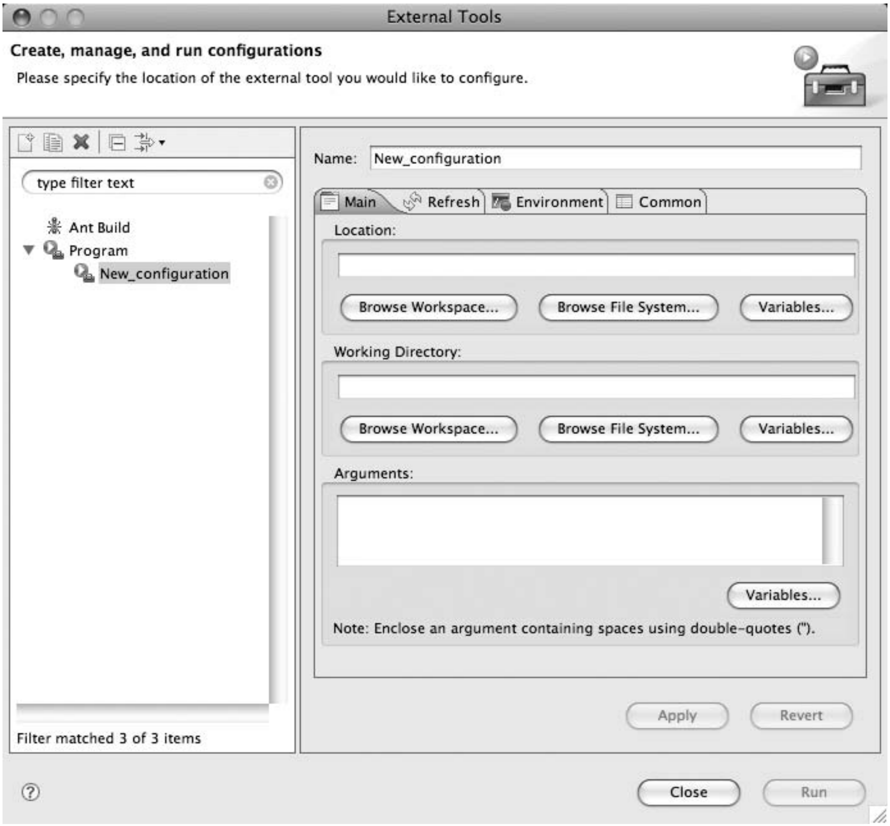


Figure 6-12. *Defining a new external tool*

This window has strong similarities to the builder definition window. In fact, the Main pane is identical, except that the name defaults to New_configuration. In this case, the name for the new configuration will be "Run full unit test suite."

Location: This points to the Python interpreter.

Workspace Directory: This points to the top-level project directory. As with the Unit Tests builder, this is specified using the Eclipse variable `${workspace_loc:/rsreader}`.

Arguments : The arguments to Python are `./setup.py test`.

The Common tab allows several options to be specified. It is shown in Figure 6-13, with the values described in the following list selected.

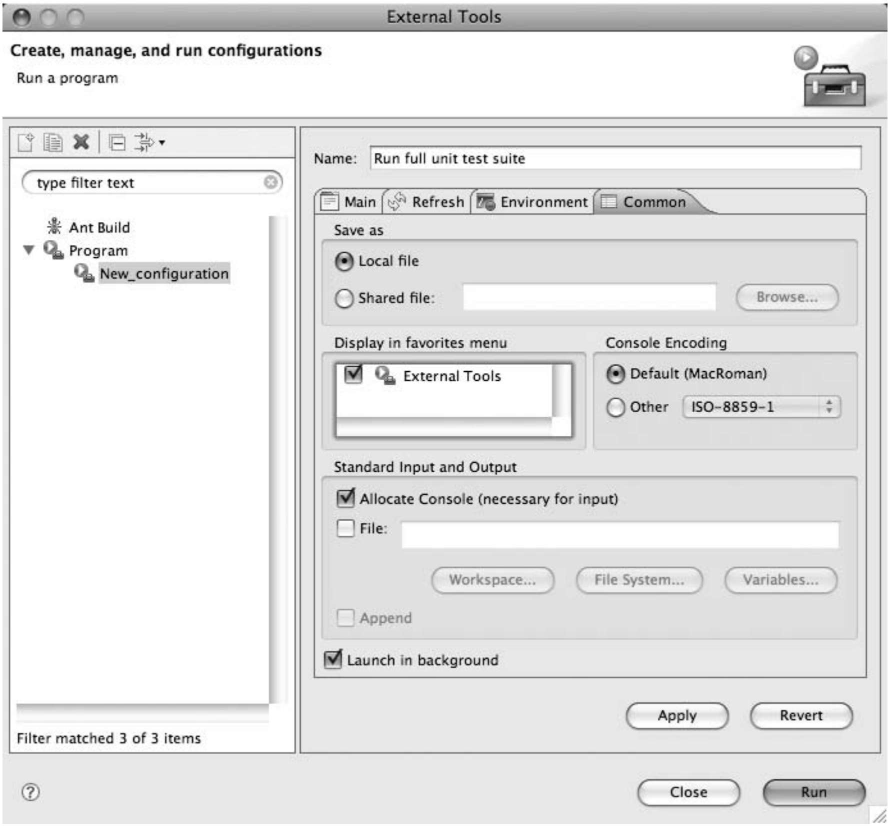


Figure 6-13. *The Common tab for the external tool*

Save as: This section determines if the tool definition is stored in the local Eclipse configuration or in the project configuration. Unless your site has a common location for the Python binary, then I advise leaving it as "Local file."

Console Encoding: This sets the console character encoding. As in Figure 6-13, the default should be left untouched.

Display in favorites menu: Checking the External Tools check box in this section places this definition into the External Tools menu. This is what we desire, so the check box is checked.

Standard Input and Output: This section chooses the input and output locations. The default allocates a console, and this is what we want.

Once the settings are chosen, you can click the Apply button, which creates the external tool entry. Nothing more remains to be done, so you can click the Close button, which will close the External Tools window and return focus to the workbench. The new task can be run from either the application menu or the external tools drop-down. On the main menu bar, the path is Run ➤ External Tools ➤ "1 Run full test suite," and from the drop-down it is just "1 Run full test suite." The output is shown in the console, as in Figure 6-14.

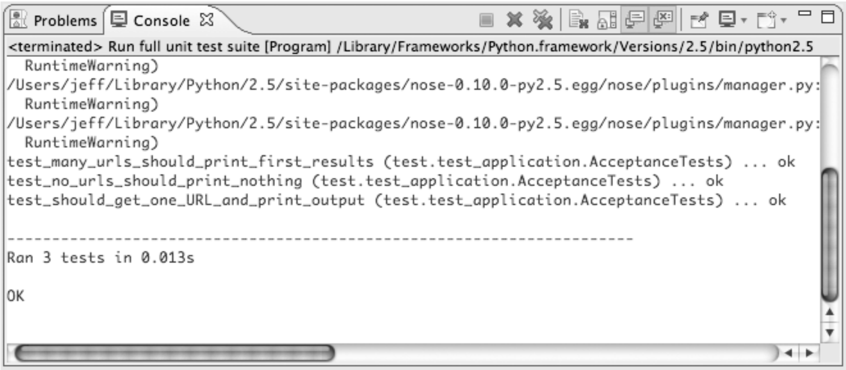


Figure 6-14. *Output of "Run full test suite"*

Buildbot with Unit Tests

The fast build suite runs automatically before each update, and the developer can run the full test suite when needed. Buildbot needs to run the full suite every time it produces a build. As with the external build task just shown, Buildbot calls `python ./setup.py test`.

Conveniently, `setup.py test` returns a meaningful exit code. On UNIX systems, it returns a zero value for success and a nonzero value for failure. Success is defined as all unit tests passing, and failure is defined as one or more unit tests failing. The `ShellCommand` build steps interpret these exit codes in the same way. The change to `master.cfg` is one line.

```
def pythonBuilder(version):
    python = python_(version)
    site_bin = site_bin_(version)
    site_pkgs = site_pkgs_(version)
    ...
    f.addStep(ShellCommand,
              command=[python, "./setup.py",  "install",
                       "--install-scripts", site_bin],
              description="Installing",
              descriptionDone="Installed")
    f.addStep(ShellCommand,
              command=[python, "./setup.py", "test"],
              description="Running unit tests",
              descriptionDone="Unit tests run")
    return f
```

The change is saved, and the Buildbot master is reconfigured:

```
$ buildbot reconfig /usr/local/buildbot/master/rsreader
```

```
sending SIGHUP to process 786
...
Reconfiguration appears to have completed successfully.
```

Committing the recent changes causes a build, or, if the changes have already been committed, then a build can be triggered from the command line. The resulting successful build is shown in Figure 6-15.

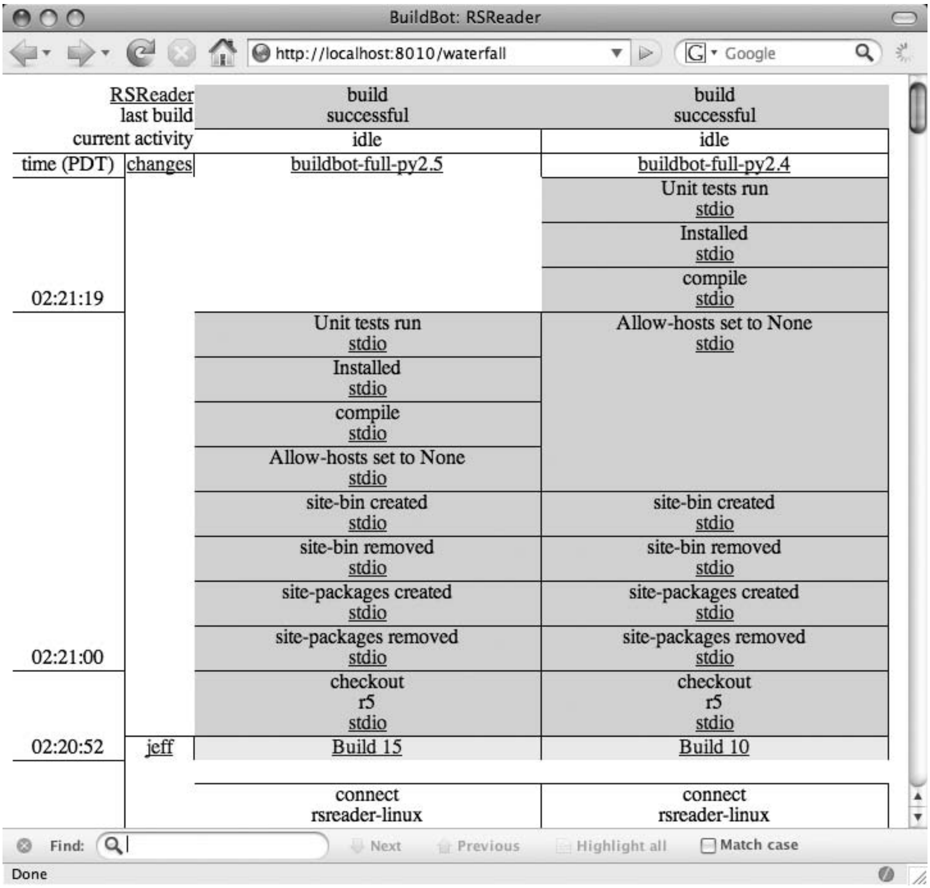


Figure 6-15. *The test step succeeds when all tests succeed.*

The build succeeds, but it must be proven to fail. You can easily do this by adding a test that always fails. You commit the change, it triggers a build, and the build fails. The unsuccessful build is shown in Figure 6-16.

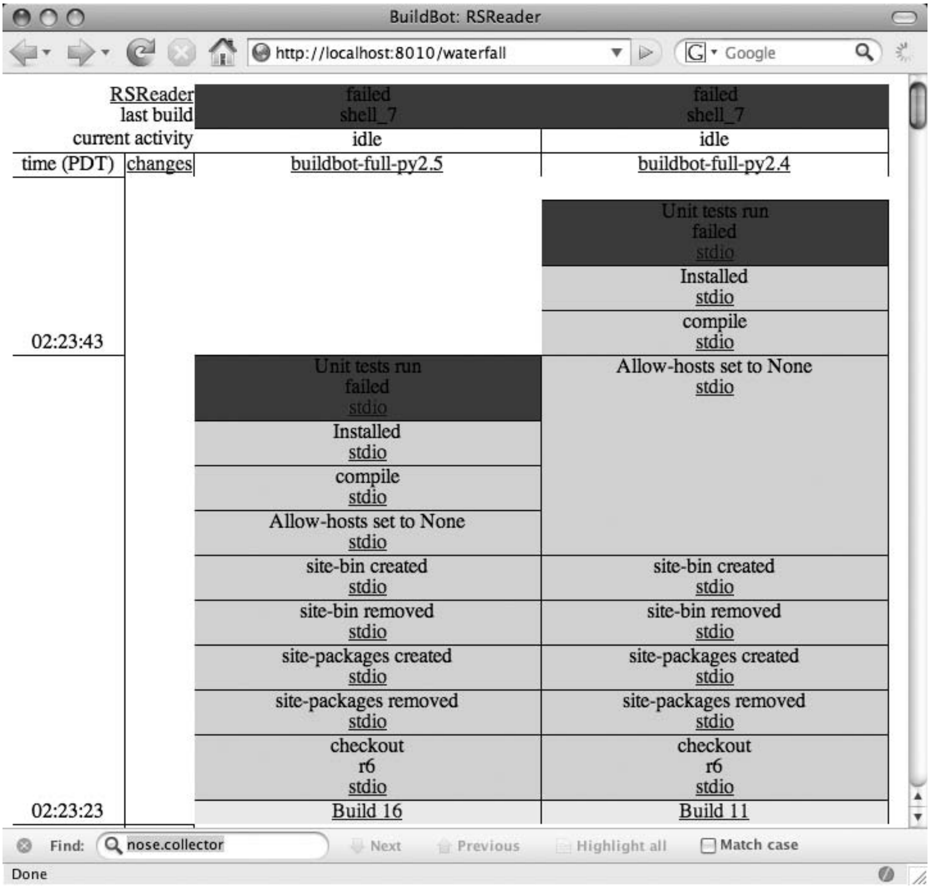


Figure 6-16. *The test step fails when a unit test fails.*

Summary

Unit tests verify the behavior of the tested code. They ensure that the code works as the programmer expects. This distinguishes them from customer or acceptance tests that determine if the code works as the user expects. Unit tests are written by programmers and for programmers. They serve as living documentation that can be programmatically verified. Concrete unit-testing benefits include fewer bugs, less debugging, live documentation, increased confidence in refactoring, and better designs on small scales.

Test-driven-development (TDD) is a collection of self-reinforcing practices. Tests are written before the code. Tests and code are written in very short cycles—sometimes just a single line of code. Only enough code is written to make a test pass, and no code goes to production without associated tests. The unit tests are executed automatically, and the build fails unless all tests succeed. The code base is constantly refactored, and there are no restrictions on where those refactorings lead. Finally, when the product ships, the tests ship with it.

Designs should be driven by the customer. Minimalist communication methods should be used to specify designs, and interaction with the on-site customer should be emphasized.

Unit tests have a common structure. They are based around assertions that report whether a test has succeeded or failed. The tests themselves have four parts: fixture setup, execution, result verification, and fixture tear-down.

Two common Python frameworks for writing, executing, and reporting unit tests are unittest and Nose. unittest is a stock Python package modeled on Smalltalk's xUnit testing framework. Nose is very good at unit test discovery and execution. It is quite extensible, and it knows how to run unittest tests. It is the basis for automating unit test execution. In the development environment, it can be run after every change using Eclipse external builders to execute only fast-running tests. It takes only a single line to connect Nose to Setuptools's testing facility, and this in turn allows the tests to be run from Buildbot.

Attributes can be used to label test methods, and Nose can use those attributes to drive test execution. One very useful example is skipping slow tests. This is often done when running in the local development environment. Setting attributes can be simplified by using Python decorators.

The next chapter is very much a continuation of this one. It fleshes out the RSReader application using TDD techniques, and focuses on the process of writing a program using TDD and refactoring.

CHAPTER 7

Test-Driven Development and Impostors

The previous chapter looked at the tools supporting TDD, but said little about TDD itself. This chapter will use several lengthy examples to show how tests are written, and along the way, you'll get to see how refactorings are performed. We'll also take a quick look at IDE refactoring support.

A consistent theme is code isolation through impostors. *Impostors*, or *test doubles*, are among the most powerful unit-testing techniques available. There is a strong temptation to overuse them, but this can result in overspecified tests.

Impostors are painful to produce by hand, but there are several packages that minimize this pain. Most fall into one of two categories based on how expectations are specified. One uses a domain-specific language, and the other uses a record-replay model. I examine a representative from each camp: pMock and PyMock.

We'll examine these packages in detail in the second half of the chapter, which presents two substantial examples. The same code will be implemented with pMock in the first example and with PyMock in the second example. Along the way, I'll discuss a few tests and demonstrate a few refactorings.[1] Each package imbues the resulting code with a distinct character, and we'll explore these effects.

Moving Beyond Acceptance Tests

Currently, all the logic for the reader application resides within the `main()` method. That's OK, though, because it's all a sham anyway. Iterative design methods focus on taking whatever functional or semifunctional code you have and fleshing it out a little more. The process continues until at some point the code no longer perpetrates a sham, and it stands on its own.

The `main()` method is a hook between Setuptools and our application class. Currently, there is no application class, so what little exists is contained in this method. The next steps create the application class and move the logic from `main()`.

Where do the new tests go? If they're application tests, then they should go into `test_application.py`. However, this file already contains a number of acceptance tests.

1. Here, I'm using the word *few* in a strictly mathematical sense. That is to say that it's smaller than the set of integers. Since there can be only *zero*, *one*, or *many* items, it follows that *many* is larger than the integers. Therefore, *few* is smaller than *many* (for all the good that does anyone).

The two should be separate, so the existing file should be copied to `acceptance_tests.py`. From Eclipse, this is done by selecting `test_application.py` in an explorer view, and then choosing Team ➤ Copy To from the context menu. From the command line, it is copied with `svn copy`.

The application tests are implemented as native Nose tests. Nose tests are functions within a module (a.k.a. a `.py` file). The test modules import assertions from the package `nose.tools`:

```
from nose.tools import *

'''Test the application class RSReader'''
```

The new application class is named `rsreader.application.RSReader`. You move the functionality from `rsreader.application.main` into `rsreader.application.RSReader.main`. At this point, you don't need to create any tests, since the code is a direct product of refactoring. The file `test_application.py` becomes nothing more than a holder for your unwritten tests.

This class RSReader has the single method `main()`. The application's `main()` function creates an instance of RSReader and then delegates it to the instance's `main(argv)` method:

```
def main():
    RSReader().main(sys.argv)


class RSReader(object):

    xkcd_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""

    def main(self, argv):
        if argv[1:]:
            print self.xkcd_items
```

The program outputs one line for each RSS item. The line contains the item's date, the feed's title, and the item's title. This is a neatly sized functional chunk. It is one action, and it has a well-defined input and output. The test assertion looks something like this:

```
assert_equals(expected_line, computed_line)
```

You should hard-code the expectation. It's already been done in the acceptance tests, so you can lift it verbatim from there.

```
expected_line = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python"""
assert_equals(expected_line, computed_line)
```

The method `listing_from_item(item, feed)` computes the `expected_line`. It uses the date, feed name, and comic title. You could pass these in directly, but that would expose the inner workings of the method to the callers.

```
expected_line = \
    """Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python"""
computed_line = RSReader().listing_from_item(item, feed)
assert_equals(expected_line, computed_line)
```

So what do items and feeds look like? The values will be coming from FeedParser. As recounted in Chapter 6, they're both dictionaries.

```
expected_line = \
    """Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python"""
item = {'date': "Wed, 05 Dec 2007 05:00:00 -0000",
        'title': "Python"}
feed = {'feed': {'title': "xkcd.com"}}
computed_line = RSReader().listing_from_item(feed, title)
assert_equals(expected_line, computed_line)
```

This shows the structure of the test, but it ignores the surrounding module. Here is the listing in its larger context:

```
from nose.tools import *

from rsreader.application import RSReader

'''Test the application class RSReader'''

def test_listing_from_item():
    expected_line = \
    """Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python"""
    item = {'date': "Wed, 05 Dec 2007 05:00:00 -0000",
            'title': "Python"}
    feed = {'feed': {'title': "xkcd.com"}}
    computed_line = RSReader().listing_from_item(feed, title)
    assert_equals(expected_line, computed_line)
```

The method `list_from_item()` hasn't been defined yet. When you run the test, it fails with an error indicating this. The interesting part of the error message is the following:

```
test.test_application.test_list_from_item ... ERROR

======================================================================
ERROR: test.test_application.test_list_from_item
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/jeff/Library/Python/2.5/site-packages/➥
nose-0.10.0-py2.5.egg/nose/case.py", line 202, in runTest
    self.test(*self.arg)
  File "/Users/jeff/Documents/ws/rsreader/src/test/test_application.py",➥
 line 13, in test_listing_from_item
```

```
    computed_line = RSReader().listing_from_item(item, feed)
AttributeError: 'RSReader' object has no attribute 'listing_from_item'

----------------------------------------------------------------------
Ran 4 tests in 0.003s

FAILED (errors=1)
```

This technique is called *relying on the compiler*. The compiler often knows what is wrong, and running the tests gives it an opportunity to check the application. Following the compiler's suggestion, you define the method missing from `application.py`:

```
def listing_from_item(self, feed, item):
    return None
```

The test runs to completion this time, but it fails:

```
FAIL: test.test_application.test_listing_from_item
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/jeff/Library/Python/2.5/site-packages/➥
nose-0.10.0-py2.5.egg/nose/case.py", line 202, in runTest
    self.test(*self.arg)
  File "/Users/jeff/Documents/ws/rsreader/src/test/test_application.py", ➥
line 16, in test_listing_from_item
    assert_equals(expected_line, computed_line)
AssertionError: 'Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python' != None

----------------------------------------------------------------------
```

Now that you're sure the test is checking the right thing, you can write the method body:

```
def list_from_item(self, feed, item):
    subst = (item['date'], feed['feed']['title'], item['title'])
    return "%s: %s: %s" % subst
```

When you run the test again, it succeeds:

```
test_many_urls_should_print_first_results➥
 (test.acceptance_tests.AcceptanceTests) ... ok
test_no_urls_should_print_nothing➥
 (test.acceptance_tests.AcceptanceTests) ... ok
test_should_get_one_URL_and_print_output➥
 (test.acceptance_tests.AcceptanceTests) ... ok
test.test_application.testing_list_from_item ... ok
```

```
----------------------------------------------------------------------
Ran 4 tests in 0.002s

OK
```

The description of this process takes two pages and several minutes to read. It seems to be a great deal of work, but actually performing it takes a matter of seconds. At the end, there is a well-tested function running in isolation from the rest of the system.

What needs to be done next? The output from all the items in the feed needs to be combined. You need to know what this output will look like. You've already defined this in `acceptance_tests.py`:

```
    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
```

Whenever possible, the same test data should be used. When requirements change, the test data is likely to change. Every location with unique test data will need to be modified independently, and each change is an opportunity to introduce new errors.

This time, you'll build up the test as in the previous example. This is the last time that I'll work through this process in so much detail. The assertion in this test is nearly identical to the one in the previous test:

```
    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    assert_equals(printed_items, computed_items)
```

The function computes the `printed_items` from the feed and the feed's items. The list of items is directly accessible from the feed object, so it is the only thing that needs to be passed in. The name that immediately comes to my mind is `feed_listing()`. The test line is as follows:

```
    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    computed_items = RSReader().feed_listing(feed)
    assert_equals(printed_items, computed_items)
```

The feed has two items. The items are indexed by the key `'entries'`:

```
    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
              'title': "Python"},
             {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
              'title': "Far Away"}]
    feed = {'feed': {'title': "xkcd.com"}, 'entries': items}
    computed_items = RSReader().feed_listing(feed)
    assert_equals(printed_items, computed_items)
```

Here's the whole test function:

```
def test_feed_listing(self):
    printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
    items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
              'title': "Python"},
             {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
              'title': "Far Away"}]
    feed = {'feed': {'title': "xkcd.com"}, 'entries': items}
    computed_items = RSReader().feed_listing(feed)
    assert_equals(printed_items, computed_items)
```

When you run the test, it complains that the method `feed_listing()` isn't defined. That's OK, though—that's what the compiler is for. However, if you're using Eclipse and Pydev, then you don't have to depend on the compiler for this feedback. The editor window will show a red stop sign in the left margin. Defining the missing method and then saving the change will make this go away.

The first definition you supply for `feed_listing()` should cause the assertion to fail. This proves that the test catches erroneous results.

```
def feed_listing(self, feed):
     return None
```

Running the test again results in a failure rather than an error, so you now know that the test works. Now you can create a successful definition. The simplest possible implementation returns a string constant. That constant is already defined: `xkcd_items`.

```
def feed_listing(self, feed):
    return self.xkcd_items
```

Now run the test again, and it should succeed. Now that it works, you can fill in the body with a more general implementation:

```
def feed_listing(self, feed):
    item_listings = [self.listing_for_item(feed, x) for x
                     in feed['entries']]
    return "\n".join(item_listings)
```

When I ran this test on my system, it succeeded. However, there was an error. Several minutes after I submitted this change, I received a failure notice from my Windows Buildbot (which I set up while you weren't looking). The error indicates that the line separator is wrong on the Windows system. There, the value is `\r\n` rather than the `\n` used on UNIX systems. The solution is to use `os.linesep` instead of a hard-coded value:

```
import os
...
def feed_listing(self, feed):
    item_listings = [self.listing_for_item(feed, x) for x
```

```
                 in feed['entries']]
    return os.linesep.join(item_listings)
```

At this point, you'll notice several things—there's a fair bit of duplication in the test data:

- xkcd_items is present in both acceptance_tests.py and application_tests.py.
- The feed items are partially duplicated in both application tests.
- The feed definition is partially duplicated in both application tests.
- The output data is partially duplicated in both tests.

As it stands, any changes in the expected results will require changes in each test function. Indeed, a change in the expected output will require changes not only in multiple functions, but in multiple files. Any changes in the data structure's input will also require changes in each test function.

In the first step, you'll extract the test data from test_feed_listing:

```
printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""

def test_feed_listing(self):
    items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
              'title': "Python"},
             {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
              'title': "Far Away"}]
    feed = {'feed': {'title': "xkcd.com"}, 'entries': items}
    computed_items = RSReader().feed_listing(feed)
    assert_equals(printed_items, computed_items)
```

You save the change and run the test, and it should succeed. The line defining printed_items is identical in both acceptance_tests.py and application_tests.py, so the definition can and should be moved to a common location. That module will be test.shared_data:

```
$ ls tests -Fa
```

```
__init__.py      acceptance_tests.pyc      application_tests.pyc
__init__.pyc      acceptance_tests.py      application_tests.py
shared_data.py
```

```
$ cat shared_data.py
```

```
"""Data common to both acceptance tests and application tests"""

__all__ = ['printed_items']
```

```
printed_items = \
"""Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away"""
```

```
$ cat acceptance_tests.py
```

```
...
from tests.shared_data import *
...
```

```
$ cat application_tests.py
```

```
...
from tests.shared_data import *
...
```

The `__all__` definition explicitly defines the module's exports. This prevents extraneous definitions from polluting client namespaces when wildcard imports are performed. Declaring this attribute is considered to be a polite Python programming practice.

The refactoring performed here is called *triangulation*. It is a method for creating shared code. A common implementation is not created at the outset. Instead, the code performing similar functions is added in both places. Both implementations are rewritten to be identical, and this precisely duplicated code is then extracted from both locations and placed into a new definition.

This sidesteps the ambiguity of what the common code might be by providing a concrete demonstration. If the common code couldn't be extracted, then it would have been a waste of time to try to identify it at the outset.

The test `test_listing_for_item` uses a subset of `printed_items`. This function tests individual lines of output, so it's used to break the `printed_items` list into a list of strings:

```
expected_items = [
    "Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python",
    "Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away""",
]
printed_items = os.linesep.join(item_listings)
```

You save the change to `shared_data.py`, run the tests, and the tests succeed. This verifies that the data used in `test_feed_listing()` has not changed. Now that the data is in a more useful form, you can change the references within `test_listing_for_item()`. You remove the definition, and the assertion now uses `expected_items`.

```
def test_listing_from_item():
    item = {'date': "Wed, 05 Dec 2007 05:00:00 -0000",
            'title': "Python"}
    feed = {'title': "xkcd.com"}
    computed_line = RSReader().listing_from_item(feed, item)
    assert_equals(expected_items[0], computed_line)
```

You run the test, and it succeeds. The expectations have been refactored, so it is now time to move on to the test fixtures. The values needed by `test_listing_from_item()` are already defined in `test_feed_listing()`, so you'll extract them from that function and remove them from the other:

```
from tests.shared_data import *

items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
          'title': "Python"},
         {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
          'title': "Far Away"}]
feed = {'feed': {'title': "xkcd.com"}, 'entries': items}

def test_feed_listing(self):
    computed_items = RSReader().feed_listing(feed)
    assert_equals(printed_items, computed_items)

def test_listing_from_item():
    computed_line = RSReader().listing_from_item(feed, items[0])
    assert_equals(expected_items[0], computed_line)
```

Renaming

Looking over the tests, it seems that there is still at least one smell. The name `printed_items` isn't exactly accurate. It's the expected output from reading xkcd, so `xkcd_output` is a more accurate name. This will mandate changes in several locations, but this process is about to become much less onerous. The important thing for the names is that they are consistent.

Inaccurate or awkward names are anathema. They make it hard to communicate and reason about the code. Each new term is a new definition to learn. Whenever a reader encounters a new definition, she has to figure out what it really means. That breaks the flow, so the more inconsistent the terminology, the more difficult it is to review the code. Readability is vital, so it is important to correct misleading names.

Traditionally, this has been difficult. Defective names are scattered about the code base. It helps if the code is loosely coupled, as this limits the scope of the changes; unit tests help to ensure that the changes are valid, too, but neither does anything to reduce the drudgery of find-and-replace. This is another area where IDEs shine.

Pydev understands the structure of the code. It can tell the difference between a function `foo` and an attribute `foo`. It can distinguish between method `foo` in class `X` and method `foo` in class `Y`, too. This means that it can rename intelligently.

This capability is available from the refactoring menu, which is available from either the main menu bar or the context menu. To rename a program element, you select its text in an editor. In this case, you're renaming the variable `printed_items`. From the main menu bar, select Refactoring ➤ Rename. (It's the same from the context menu.) There are also keyboard accelerators available for this, and they're useful to know.

Choosing the Rename menu item brings up the window shown in Figure 7-1. Enter the new name **xkcd_output**.



Figure 7-1. *The Rename refactoring window*

At this point, you can preview the changes by clicking the Preview button. This brings up the refactoring preview window shown in Figure 7-2.

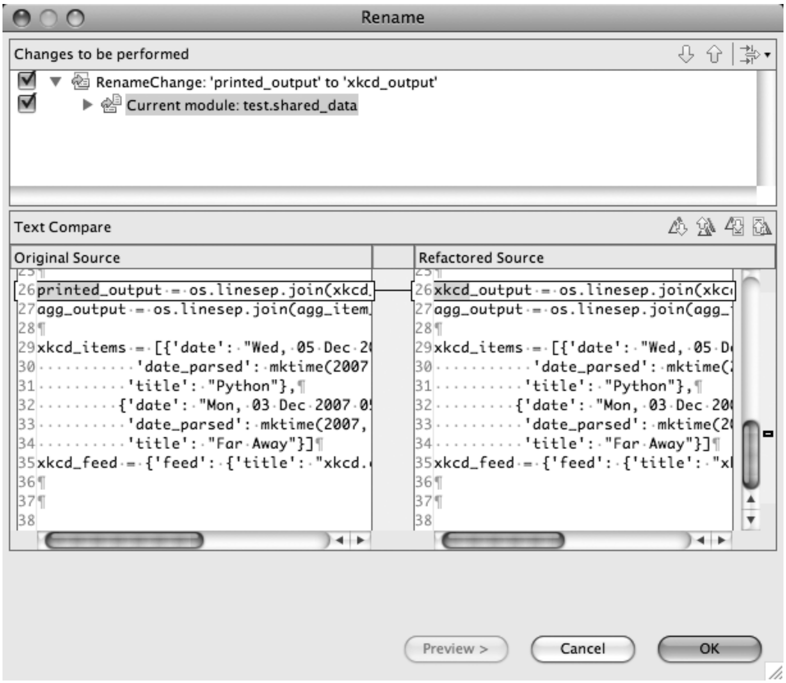


Figure 7-2. *The refactoring preview window*

Each candidate refactoring can be viewed and independently selected or unselected though the check box to its left. Pydev not only checks the code proper, but it checks string literals and comments, too, so the preview is often a necessary step, even with simple renames.

I find it edifying to see how many places the refactoring touches the program. It reminds me how the refactored code is distributed throughout the program, and it conveys an impression of how tightly coupled the code is.

When satisfied, click OK, and the refactoring will proceed. After a few seconds, the selected changes will be complete.

Overriding Existing Methods: Monkeypatching

The code turning a feed object into text has been written. The next step converts URLs into feed objects. This is the magic that FeedParser provides. The test harness doesn't have control over network connections, and the Net at large can't be controlled without some pretty involved network hackery. More important, the tests shouldn't be dependent on external resources unless they're included as part of the build.

All of these concerns can be surmounted by hacking FeedParser on the fly. Its parse routine is temporarily replaced with a function that behaves as desired. The test is defined first:

```
def test_feed_from_url():
    url = "http://www.xkcd.com/rss.xml"
    assert_equals(feed, RSReader().feed_from_url(url))
```

The test method runs, and it fails with an error stating that `feed_from_url()` has not been defined. The method is defined as follows:

```
def feed_from_url(self, url):
    return None
```

The test is run, and fails with a message indicating that `feed` does match the results returned from `feed_from_url()`. Now for the fun stuff. A fake parse method is defined in the test, and it is hooked into FeedParser. Before this is done, the real parse method is saved, and after the test completes, the saved copy is restored.

```
import feedparser
...
def test_feed_from_url():
   url = "http://www.xkcd.com/rss.xml"
    def  parse_stub(url): # define stub
        return feed
    real_parse = feedparser.parse # save real value
    feedparser.parse = parse_stub # attach stub
    try:
        assert_equals(feed, RSReader().feed_from_url(url))
    finally:
        feedparser.parse = real_parse # restore real value
```

The test is run, and it fails in the same manner as before. Now the method is fleshed in:

```
import feedparser
...
    def feed_from_url(self, url):
        return feedparser.parse(url)
```

The test runs, and it succeeds.

Monkeypatching and Imports

In order for monkeypatching to work, the object overridden in the test case and the object called from the subject must refer to the same object. This generally means one of two things. If the subject imports the module containing the object to be overridden, then the test must do the same. This is illustrated in Figure 7-3. If the subject imports the overridden object from the module, then the test must import the subject module, and the reference in the subject module must be overridden. This is reflected in Figure 7-4.

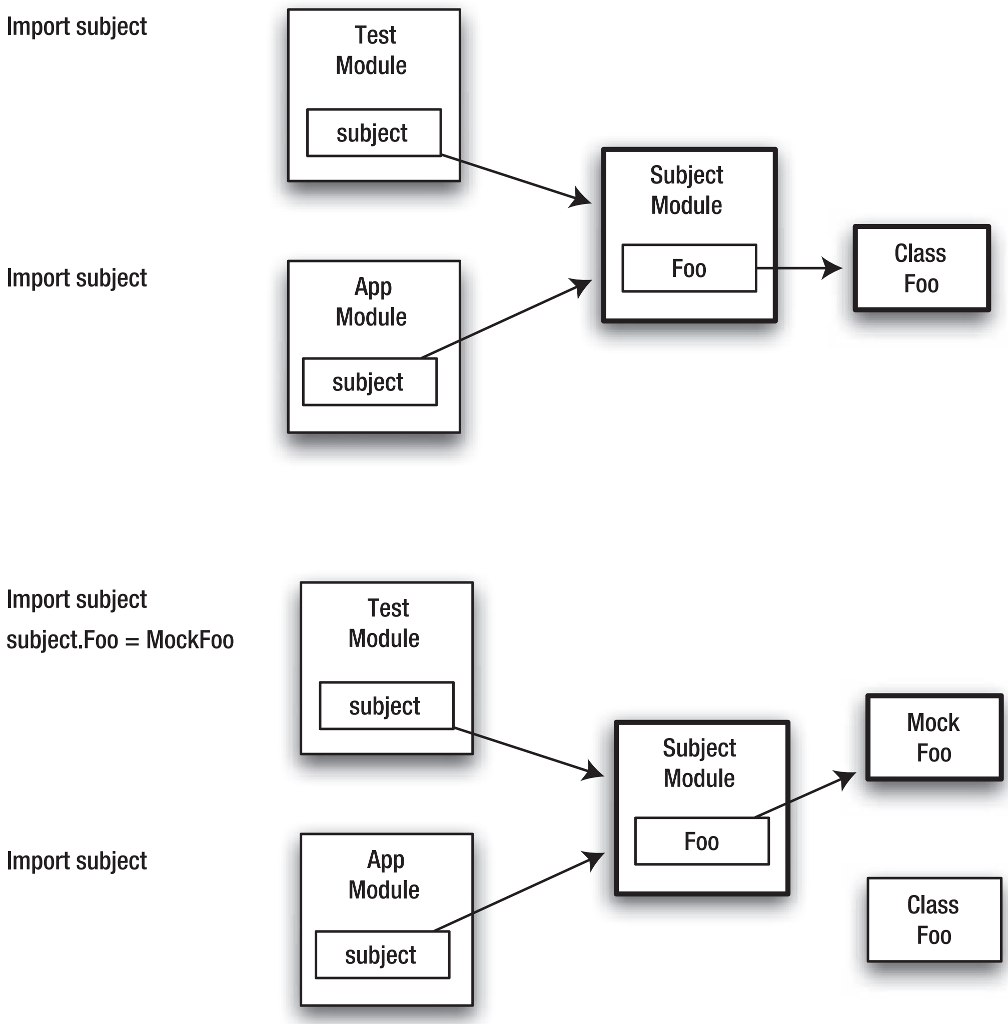


Figure 7-3. *Replacing an object when the subject imports the entire module containing the object*

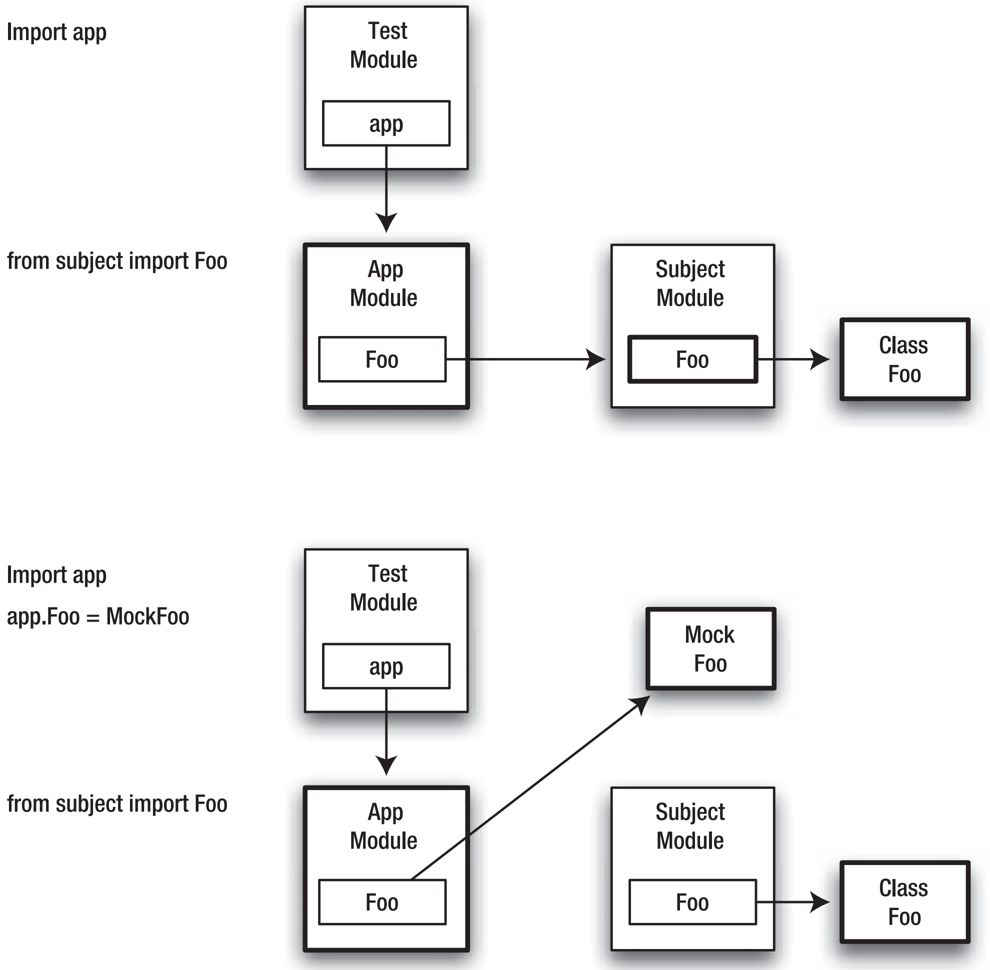


Figure 7-4. *Replacing an object when the subject directly imports the object*

It is tempting to import the subject module directly into the test's namespace. However, this does not work. Altering the test's reference doesn't alter the subject's reference. It results in the situation shown in Figure 7-5, where the test points to the mock, but the rest of the code still points to the real object. This is why it is necessary to alter the reference to the subject module, as in Figure 7-4.

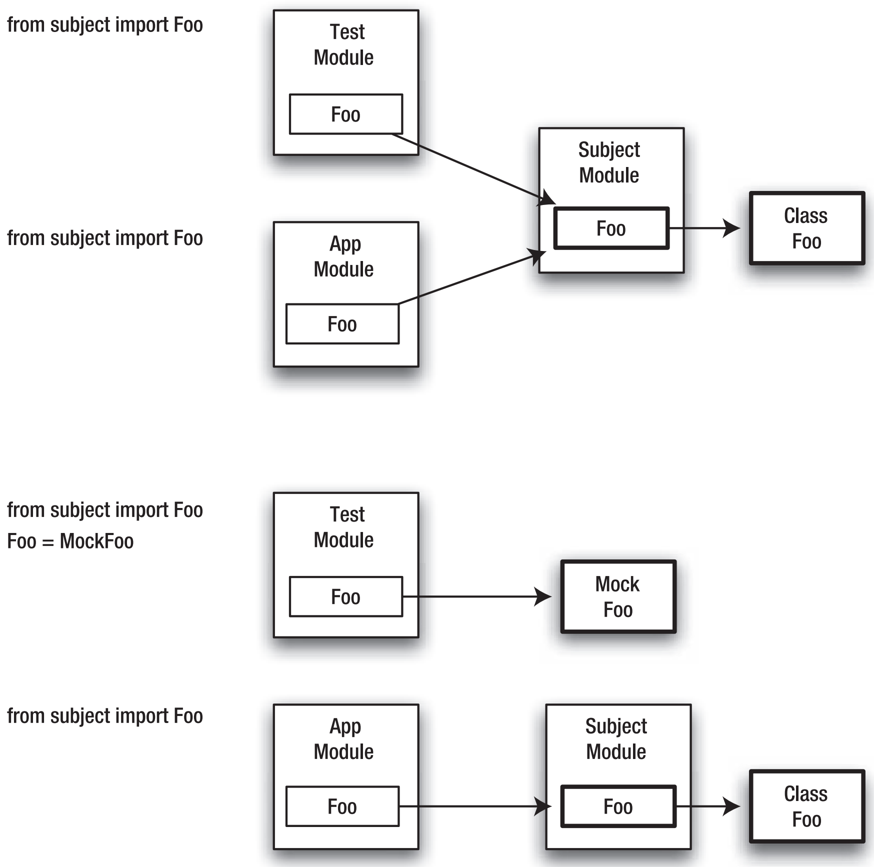


Figure 7-5. *Why replacing an object imported directly into the test's namespace doesn't work*

The Changes Go Live

At this point, URLs can be turned into feeds, and feeds can be turned into output. Everything is available to make a working application. The new `main()` method is as follows:

```
def main(self, argv):
    if argv[1:]:
        url = argv[1]
        print self.listing_from_feed(self.feed_from_url(url))
```

The test suite is run, and the acceptance tests fail:

```
test_many_urls_should_print_first_results➥
(test.acceptance_tests.AcceptanceTests) ... FAIL
test_no_urls_should_print_nothing (test.acceptance_tests.AcceptanceTests) ... ok
test_should_get_one_URL_and_print_output (test.acceptance_tests.AcceptanceTests)➥
 ... FAIL
```

```
test.test_application.test_list_from_item ... ok
test.test_application.test_list_from_feed ... ok
test.test_application.test_list_from_url ... ok
test.test_application.test_feed_from_url ... ok

...

======================================================================
FAIL: test_should_get_one_URL_and_print_output➥
(test.acceptance_tests.AcceptanceTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/jeff/Documents/ws/rsreader/src/test/acceptance_tests.py",➥
line 25, in test_should_get_one_URL_and_print_output
    self.assertStdoutEquals(self.printed_items + "\n")
  File "/Users/jeff/Documents/ws/rsreader/src/test/acceptance_tests.py",➥
line 38, in assertStdoutEquals
    self.assertEquals(expected_output, sys.stdout.getvalue())
AssertionError: 'Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com:➥
Python\nMon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away\n' != '\n'

----------------------------------------------------------------------
Ran 7 tests in 0.015s

FAILED (failures=2)
```

The acceptance tests are now using real code, and the test cases have a problem.

Using Data Files

The failing tests are trying to access the file `xkcd.rss.xml`. This file doesn't exist, so the code is dying. These files should contain real RSS data that has been trimmed down to produce the expected results. I've done this already. You can simply download the file from `www.theblobshop.com/famip/xkcd.rss.xml` to a new directory, `src/test/data`.

With this file in place, the tests still fail. The acceptance tests need to specify the full path to the data file. The path is relative to the test being run, so it can be extracted from the test module's `__file__` attribute:

```
import StringIO
import sys
from unittest import TestCase

from test.shared_data import *
from rsreader.application import main

module = sys.modules[__name__]
this_dir = os.path.dirname(os.path.abspath(module.__file__))
```

```
xkcd_rss_xml = os.path.join(this_dir, 'data', 'xkcd.rss.xml')

class AcceptanceTests(TestCase):

    def setUp(self):
        self.old_value_of_stdout = sys.stdout
        sys.stdout = StringIO.StringIO()
        self.old_value_of_argv = sys.argv

    def tearDown(self):
        sys.stdout = self.old_value_of_stdout
        sys.argv = self.old_value_of_argv

    def test_should_get_one_URL_and_print_output(self):
        sys.argv = ["unused_prog_name", xkcd_rss_xml]
        main()
        self.assertStdoutEquals(expected_output + "\n")

    def test_no_urls_should_print_nothing(self):
        sys.argv = ["unused_prog_name"]
        main()
        self.assertStdoutEquals("")

    def test_many_urls_should_print_first_results(self):
        sys.argv = ["unused_prog_name", xkcd_rss_xml, "excess"]
        main()
        self.assertStdoutEquals(expected_output + "\n")

    def assertStdoutEquals(self, expected_output):
        self.assertEquals(expected_output, sys.stdout.getvalue())
```

With this change in place, the tests run, and they all succeed. The first pass at the application is complete. It can be installed and run from the command line.

Isolation

Isolating the components under test from the system at large is a major theme in unit testing. You've seen this with the method `feed_from_url()`. It has a dependency upon the function `feedparser.parse()` that was temporarily separated by the replacement of the function with a fake implementation.

These dependencies come in three main forms:

Dependencies can be introduced to functions and methods as arguments: In the function call `f(x)`, the function depends upon `x`. The object may be passed as an argument to other callables, methods may be invoked upon it, it may be returned, it may be raised as an exception, or it may be captured. When captured, it may be assigned to a variable or an attribute or bundled into a closure.

Dependencies can be introduced as calls to global entities: Global entities include packages, classes, and functions. In languages such as C and Java, these are static declarations, to some extent corresponding to the type system. In Python, these are much more dynamic. They're first-class objects that are not only referenced through the global namespace—they can be introduced through arguments as well. The method `f(x)` introduces a dependency on the package `os`:

```
def f(filename):
    x = os.listdir(filename)
```

Dependencies can be introduced indirectly: They are introduced as the return values from functions and methods, as exceptions, and as values retrieved from attributes. These are in some sense the product of the first two dependency classes. Modeling these is inherent in accurately modeling the first.

To test in isolation, these dependencies must be broken. Choosing an appropriate design is the best way to do this. The number of objects passed in as arguments should be restricted. The number of globals accessed should be restricted, too, and as little should be done with return values as possible. Even more important, side effects (assignments) should be restricted as much as possible. However, coupling is inescapable. A class with no dependencies and no interactions rarely does anything of interest.

The remaining dependencies are severed through a set of techniques known as mocking. *Mocking* seeks to replace the external dependencies with an impostor. The impostor has just enough functionality to allow the tested unit to function. Impostors perform a subset of these functions:

- Fulfilling arguments
- Avoiding references to objects outside the unit
- Tracking call arguments
- Forcing return values
- Forcing exceptions
- Verifying that calls were made
- Verifying call ordering

There are four categories of impersonators:

Dummies: These are minimal objects. They are created so that the system as a whole will run. They're important in statically typed languages. An accessor method may store a derived class, but no such class exists in the section of the code base under examination. A class is derived from the abstract base class, and the required abstract methods are created, but they do nothing, and they are never called in the test. This allows the tests to compile. In Python, it is more common to see these in the case of conditional execution. An argument may only be used in one branch of the conditional. If the test doesn't exercise that path, then it passes a dummy in that argument, and the dummy is never used in the test.

Stubs: These are more substantial than dummies. They implement minimal behavior. In stubs, the results of a call are hard-coded or limited to a few choices. The isolation of `feed_from_url()` is a clear-cut example of this. The arguments weren't checked, and there weren't any assertions about how often the method stub was called, not even to ensure that it was called at all. Implementing any of this behavior requires coding.

Mocks: These are like stubs, but they keep track of expectations and verify that they were met. Different arguments can produce different outcomes. Assertions are made about the calls performed, the arguments passed, and how often those calls are performed, or even if they are performed at all. Values returned or exceptions raised are tracked, too. Performing all of this by hand is involved, so many mock object frameworks have been created, with most using a concise declarative notation.

Fakes: These are more expansive and often more substantial than mock objects. They are typically used to replace a resource-intensive or expansive subsystem. The subsystem might use vast amounts of memory or time, with time typically being the important factor for testing. The subsystem might be expansive in the sense that it depends on external resources such as a network-locking service, an external web service, or a printer. A database is the archetypical example of a faked service.

Rolling Your Own

Dummies are trivial to write in Python. Since Python doesn't check types, an arbitrary string or numeric value suffices in many cases.

In some cases, you'll want to add a small amount of functionality to an existing class or override existing functionality with dummies or stubs. In this case, the test code can create a subclass of the subject. This is commonly done when testing a base class. It takes little effort, but Python has another way of temporarily overriding functionality, which was shown earlier.

Monkeypatching takes an existing package, class, or callable, and temporarily replaces it with an impostor. When the test completes, the monkeypatch is removed. This approach isn't easy in most statically typed languages, but it's nearly trivial in Python. With instances created as test fixtures, it is not necessary to restore the monkeypatch, since the change will be lost once an individual test completes. Packages and classes are different, though. Changes to these persist across test cases, so the old functionality must be restored.

There are several drawbacks to monkeypatching by hand. Undoing the patches requires tracking state, and the problem isn't straightforward—particularly when properties or subclasses are involved. The changes themselves require constructing the impostor, so this piles up difficulties.

Hand-coding mocks is involved. Doing one-offs produces a huge amount of ugly setup code. The logic within a mocked method becomes tortuous. The mocks end up with a mishmash of methods mapping method arguments to output values. At the same time, this interacts with method invocation counting and verification of method execution. Any attempt to really address the issues in a general way takes you halfway toward creating a mock object package, and there are already plenty of those out there. It takes far less time to learn how to use the existing ones than to write one of your own.

Python Quirks

In languages such as Java and C++, subclassing tends to be preferred to monkeypatching. Although Python uses inheritance, programmers rely much more on duck typing—if it looks like a duck and quacks like a duck, then it must be a duck. Duck typing ignores the inheritance structure between classes, so it could be argued that monkeypatching is in some ways more Pythonic.

In many other languages, instance variables and methods are distinct entities. There is no way to intercept or modify assignments and access. In these languages, instance variables directly expose the implementation of a class. Accessing instance variables forever chains a caller to the implementation of a given object, and defeats polymorphism. Programmers are exhorted to access instance values through getter and setter methods.

The situation is different in Python. Attribute access can be redirected through hidden getter and setter methods, so attributes don't directly expose the underlying implementation. They can be changed at a later date without affecting client code, so in Python, attributes are valid interface elements.

Python also has operator overloading. Operator overloading maps special syntactic features onto underlying functions. In Python, array element access maps to the function `__getitem__()`, and addition maps to the method `__add__()`. More often than not, modern languages have some mechanism to accomplish this magic, with Java being a dogmatic exception.

Python takes this one step further with the concept of protocols. *Protocols* are sequences of special methods that are invoked to implement linguistic features. These include generators, the `with` statement, and comparisons. Many of Python's more interesting linguistic constructions can be mocked by understanding these protocols.

Mocking Libraries

Mocking libraries vary in the features they provide. The method of mock construction is the biggest discriminator. Some packages use a domain-specific language, while others use a record-playback model.

Domain-specific languages (DSLs) are very expressive. The constructed mocks are very easy to read. On the downside, they tend to be very verbose for mocking operator overloading and for specifying protocols. DSL-driven mock libraries generally descend from Java's jMock. It has a strong bias toward using only vanilla functions, and the descendent DSLs reflect this bias.

Record-replay was pioneered by Java's EasyMock. The test starts in a record mode, mock objects are created, and the expected calls are performed on them. These calls are recorded, the mock is put into playback mode, and the calls are played back. The approach works very well for mocking operator overloading, but its implementation is fraught with peril. Unsurprisingly, the additional work required to specify results and restrictions makes the mock setup more confusing than one might expect.

Two mocking libraries will be examined in this chapter: pMock and PyMock. *pMock* is a DSL-based mocking system. It only works on vanilla functions, and its DSL is clear and concise. Arguments to mocks may be constrained arbitrarily, and pMock has excellent failure reporting. However, it is poor at handling many Pythonic features, and monkeypatching is beyond its ken.

PyMock combines mocks, monkeypatching, attribute mocking, and generator emulation. It is primarily based on the record-replay model, with a supplementary DSL. It handles generators, properties, and magic methods. One major drawback is that its failure reports are fairly opaque.

In the next section, the example is expanded to handle multiple feeds. The process is demonstrated using first pMock, and then PyMock.

Aggregating Two Feeds

In this example, two separate feeds need to be combined, and the items in the output must be sorted by date. As with the previous example, the name of the feed should be included with its title, so the individual feed items need to identify where they come from. A session might look like this:

```
$ rsreader http://www.xkcd.com/rss.xml http://www.pvponline.com/rss.xml
```

```
Thu, 06 Dec 2007 06:00:36 +0000: PvPonline: Kringus Risen - Part 4
Wed, 05 Dec 2007 06:00:45 +0000: PvPonline: Kringus Risen - Part 3
Wed, 05 Dec 2007 05:00:00 -0000: xkcd.com: Python
Mon, 03 Dec 2007 05:00:00 -0000: xkcd.com: Far Away
```

The feeds must be retrieved separately. This is a design constraint from FeedParser. It pulls only one at a time, and there is no way to have it combine the two feeds. Even if the package were bypassed, this would still be a design constraint. The feeds must be parsed separately before they can be combined. In all cases, every feed item needs to be visited once.

The feeds could be combined incrementally, but doing things incrementally tends to be tougher than working in batches. There are multiple approaches to combining the feeds, and they all fundamentally answer the question: how do you locate the feed associated with an item?

One approach places the intelligence outside the feeds. One list aggregates either the feeds or the fed items. A dictionary maps the individual items back to their parent feeds. This can be wrapped into a single class that handles aggregation and lookup. The number of internal data structures is high, but it works.

In another approach, the FeedParser objects can be patched. A new key pointing back the parent feed is added to each entry. This involves mucking about with the internals of code belonging to third-party packages.

Creating a parallel set of data structures (or new classes) is yet another option. The interesting aspects of the aggregated feeds are modeled, and the uninteresting ones are ignored. The downsides are that we're creating a duplicate object hierarchy, and it duplicates some of the functionality in FeedParser. The upsides are that it is very easy to build using a mocking framework, and it results in precisely the objects and classes needed for the project.

What routines are needed? The method for determining this is somewhat close to pseudocode planning. Starting with a piece of paper, the new section of code is outlined, and the outline is translated into a series of tests. The list isn't complete or exhaustive—it just serves as a starting point.

```
def _pending_test_new_main():
    """Hooking in new code to main"""

def _pending_test_get_feeds_from_urls():
    """Should get a feed for every URL"""

def _pending_test_combine_feeds():
    """Should combine feeds into a list of FeedEntries"""

def _pending_test_add_single_feed():
    """Should add a single feed to a set of feeds"""

def _pending_test_create_entry():
    """Create a feed item from a feed and a feed entry"""

def _pending_test_feed_listing_is_sorted():
    """Should sort the aggregate feed listing"""

def _pending_test_feed_entry_listing():
    """Should produce a correctly formatted listing from a feed entry"""
```

The string `_pending_` prefixing each test tells those reading your code that the tests are not complete. The starting underscore tells Nose that the function is not a test. When you begin writing the test, the string `_pending_` is removed.

A Simple pMock Example

pMock is installed with the command `easy_install pmock`. It's a pure Python package, so there's no compilation necessary, and it should work on any system.

A simple test shows how to use pMock. The example will calculate a triangle's perimeter:

```
def test_perimeter():
    assert_equals(4, perimeter(triangle))
```

pMock imitates the triangle object:

```
def test_perimeter():
    triangle = Mock()
    assert_equals(4, perimeter(triangle))
```

The expected method calls `triangle.side(0)` and `triangle.side(1)` need to be modeled. They return `1` and `3`, respectively.

```
def test_perimeter():
    triangle = Mock()
    triangle.expects(once()).side(eq(0)).will(return_value(1))
    triangle.expects(once()).side(eq(1)).will(return_value(3))
    assert_equals(4, perimeter(triangle))
```

Each expectation has three parts. The `expects()` clause determines how many times the combination of method and arguments will be invoked. The second part determines the method name and argument constraints. In this case, the calls have one argument, and it must be equal to `0` or `1`. `eq()` and `same()` are the most common constraints, and they are equivalent to Python's `==` and `is` operators. The optional `will()` clause determines the method's actions. If present, the method will either return a value or raise an exception.

The simplest method fulfilling the test is the following:

```
def perimeter(triangle):
    return 4
```

When you run the test, it succeeds even though it doesn't call the triangle's `side()` methods. You must explicitly check each mock to ensure that its expectations have been met. The call `triangle.verify()` does this:

```
def test_perimeter():
    triangle = Mock()
    triangle.expects(once()).side(eq(0)).will(return_value(1))
    triangle.expects(once()).side(eq(1)).will(return_value(3))
    assert_equals(4, perimeter(triangle))
    triangle.verify()
```

Now when you run it the test, it fails. The following definition satisfies the test:

```
def perimeter(triangle):
   return triangle.side(0) + triangle.side(1)
```

Implementing with pMock

To use mock objects, there must be a way of introducing them into the tested code. There are four possible ways of doing this from a test. They can be passed in class variables, they can be assigned as instance variables, they can be passed in as arguments, or they can be introduced from the global environment.

Test: Defining combine_feeds

Mocking calls to `self` poses a problem. This could be done with monkeypatching, but that's not a feature offered by pMock. Instead, `self` is passed in as a second argument, and it introduces the mock. In this case, the auxiliary `self` is used to help aggregate the feed.

```
def test_combine_feeds():
    """"Combine one or more feeds""""
    aggregate_feed = Mock()
    feeds = [Mock(), Mock()]
    aggregate_feed.expects(once()).add_single_feed(same(feeds[0]))
    aggregate_feed.expects(once()).add_single_feed(same(feeds[1]))
    RSReader().combine_feeds(aggregate_feed, feeds)
    aggregate_feed.verify()
```

The test fails. The method definition fulfilling the test is as follows:

```
def combine_feeds(self, aggregate_feed, feeds):
    for x in feeds:
       aggregate_feed.add_single_feed(x)
```

The test now succeeds.

Test: Defining add_single_feed

The next test is `test_add_single_feed()`. It verifies that `add_single_feed()` creates an aggregate entry for each entry in the feed:

```
def test_add_singled_feed():
    """Should add a single feed to a set of feeds"""
    entries = [Mock(), Mock()]
    feed = {'entries': entries}
    aggregate_feed = Mock()
    aggregate_feed.expects(once()).create_entry(same(feed), same(entries[0]))
    aggregate_feed.expects(once()).create_entry(same(feed), same(entries[1]))
    RSReader().add_single_feed(aggregate_feed, feed)
    aggregate_feed.verify()
```

The test fails. The method `RSReader.add_single_feed()` is defined:

```
def add_single_feed(self, feed_aggregator, feed):
    for e in feed['entries']:
        feed_aggregator.create_entry(e)
```

The test now passes. There is a problem, though. The two tests have different definitions for `add_single_feed`. In the first, it is called as `add_single_feed(feed)`. In the second, it is called as `add_single_feed(aggregator_feed, feed)`. In a statically typed language, the development environment or compiler would catch this, but in Python, it is not caught. This is both a boon and a bane. The boon is that a test can completely isolate a single method call from the rest of the program. The bane is that a test suite with mismatched method definitions can run successfully.

The second test's definition is obviously the correct one, so you revise the first one. It is also apparent that the same problem will exist for `create_entry`, so you fix this expectation at the same time.

```
def test_combine_feeds():
    """"Combine one or more feeds""""
    aggregate_feed = Mock()
    feeds = [Mock(), Mock()]
    aggregate_feed.expects(once()).add_single_feed(same(aggregate_feed),
        same(feeds[0]))
    aggregate_feed.expects(once()).add_single_feed(same(aggregate_feed),
        same(feeds[1]))
    subject = RSReader().combine_feeds(aggregate_feed, feeds)
    aggregate_feed.verify()
```

```
def test_add_singled_feed():
    """Should add a single feed to a set of feeds"""
    entries = [Mock(), Mock()]
    feed = {'entries': entries}
    aggregate_feed = Mock()
    aggregate_feed.expects(once()).create_entry(same(aggregate_feed),
        same(feed), same(entries[0]))
    aggregate_feed.expects(once()).create_entry(same(aggregate_feed),
        same(feed), same(entries[1]))
    RSReader().add_single_feed(aggregate_feed, feed)
    aggregate_feed.verify()
```

And the method definitions are also changed:

```
def combine_feeds(self, feed_aggregator, feeds):
    for f in feeds:
        feed_aggregator.add_single_feed(feed_aggregator, f)

def add_single_feed(self, feed_aggregator, feed):
    for e in feed['entries']:
        feed_aggregator.create_entry(feed_aggregator, feed, e)
```

In some sense, strictly using mock objects induces a style that obviates the need for `self`. It maps very closely onto languages with multimethods. While the second copy of `self` is merely conceptually ugly in other languages, Python's explicit `self` makes it typographically ugly, too.

Refactoring: Extracting AggregateFeed

The second `self` variable serves a purpose, though. If named to reflect its usage, then it indicates which class the method belongs to. If that class doesn't exist, then it strongly suggests that it should be created. In this case, the class is `AggregateFeed`.

You create the new class, and one by one you move over the methods from `RSReader`. First you modify the test, and then you move the corresponding method. You repeat this process until all the appropriate methods have been moved.

```
from rsreader.application import AggregateFeed, RSReader
...
def test_combine_feeds():
    """Should combine feeds into a list of FeedEntries"""
    subject = AggregateFeed()
    mock_feeds = [Mock(), Mock()]
    aggregate_feed = Mock()
    aggregate_feed.expects(once()).add_single_feed(same(aggregate_feed),
                                                    same(mock_feeds[0]))
    aggregate_feed.expects(once()).add_single_feed(same(aggregate_feed),
                                                    same(mock_feeds[1]))
    subject.combine_feeds(aggregate_feed, mock_feeds)
    aggregate_feed.verify()
```

The test fails because the class AggregateFeed is not defined. The new class is defined:

```
class AggregateFeed(object):
    """Aggregates several feeds"""

    pass
```

The tests are run, and they still fail, but this time because the method AggregateFeed.combine_feeds() is not defined. The method is moved to the new class:

```
class AggregateFeed(object):
    """Aggregates several feeds"""

    def combine_feeds(self, feed_aggregator, feeds):
        for f in feeds:
            feed_aggregator.add_single_feed(feed_aggregator, f)
```

Now the test succeeds. With mock objects, methods can be moved easily between classes without breaking the entire test suite.

Refactoring: Moving add_single_feed

The process is continued with test_add_single_feed(). You alter test_add_single_feed to create AggregateFeed as the test subject:

```
def test_add_single_feed():
    """Should add a single feed to a set of feeds"""
    entries = [Mock(), Mock()]
    feed = {'entries': entries}
    aggregate_feed = Mock()
    aggregate_feed.expects(once()).create_entry(same(aggregate_feed),
        same(feed), same(entries[0]))
    aggregate_feed.expects(once()).create_entry(same(aggregate_feed),
        same(feed), same(entries[1]))
    AggregateFeed().add_single_feed(aggregate_feed, feed)
    aggregate_feed.verify()
```

The test fails. You move the method from RSReader to AggregateFeed to fix this:

```
class AggregateFeed(object):
    """Aggregates several feeds"""

    def combine_feeds(self, feed_aggregator, feeds):
        for f in feeds:
            feed_aggregator.add_single_feed(feed_aggregator, f)

    def add_single_feed(self, feed_aggregator, feed):
        for e in feed['entries']:
            feed_aggregator.create_entry(feed_aggregator, feed, e)
```

When you run the test it now succeeds.

Test: Defining create_entry

The next test is `test_create_entry()`. It takes an existing feed and an entry from that feed, and converts it to the new model. The new model has not been defined. The test assumes that it uses a factory to produce new instances. This factory is an instance variable in `AggregateFeed`. The object created by the factory is added to `aggregate_feed()`:

```
def test_create_entry():
    """Create a feed item from a feed and a feed entry"""
    agg_feed = AggregateFeed()
    agg_feed.feed_factory = Mock()
    (aggregate_feed, feed, entry, converted) = (Mock(), Mock(), Mock(), Mock())
    agg_feed.feed_factory.expects(once()).from_parsed_feed(same(feed),
        same(entry)).will(return_value(converted))
    aggregate_feed.expects(once()).add(same(converted))
    agg_feed.create_entry(aggregate_feed, feed, entry)
    aggregate_feed.verify()
```

The test fails, so you add the following code:

```
def create_entry(self, feed_aggregator, feed, entry):
    """"Create a new feed entry and aggregate it"""
    feed_aggregator.add(self.feed_factory.from_parsed_feed(feed, entry))
```

And now the test succeeds.

Test: Ensuring That AggregateFeed Creates a FeedEntry Factory

`create_entry` has given birth to three new tests:

```
def _pending_test_aggregate_feed_creates_factory():
    """Verify that the AggregateFeed object creates a factory when instantiated"""

def _pending_test_feed_entry_from_parsed_feed():
    """Factory method to create a new feed entry from a parsed feed"""

def _pending_test_add():
    """Add an a feed entry to the aggregate"""
```

Checking to see if the `AggregateFeed` creates a factory seems like the easiest test to me, so we'll tackle it first, but it does take a little consideration of the program's larger structure.

Each entry in a feed will be represented by an instance of the class `FeedEntry`. The factory could be a function or another class, but that's probably making things a little too complicated. Instead, it will be a method within `FeedEntry`.

```
from rsreader.app import AggregateFeed, FeedEntry, RSReader
...
def test_aggregate_feed_creates_factory():
    """Verify that the AggregatedFed object creates a factory
       when instantiated"""
    assert_equals(FeedEntry, AggregateFeed().feed_factory)
```

The test fails because the FeedEntry class is not defined yet.

```
class FeedEntry(object):
    """Combines elements of a feed and a feed entry.
       Allows multiple feeds to be aggregated without losing
       feed specific information."""
```

The test now runs, but fails because AggregateFeed.__init__ is not defined.

```
class AggregateFeed(object):
    """Aggregates several feeds"""

    def __init__(self):
        self.feed_factory = FeedEntry
```

The test now passes.

Test: Defining add

The next test you'll write is test_add().The add() method records the newly aggregated methods. At this point, the testing becomes very concrete.

```
from sets import Set
...
def test_add():
    """Add an a feed entry to the aggregate"""
    entry = Mock()
    subject = AggregateFeed()
    subject.add(entry)
    assert_equals(Set([entry]), subject.entries)
```

The test fails. The corresponding definition is as follows:

```
from sets import Set
...
def add(self, entry):
     self.entries = Set([entry])
```

The test passes this time. This definition is fine for a single test, but it needs to be refactored into something more useful.

Test: AggregateFeed.entries Is Always Initialized to a Set

The empty set should be defined when a feed is created. A new test ensures this:

```
def test_entries_is_always_defined():
    """The entries set should always be defined"""
    assert_equals(Set(), AggregateFeed().entries)
```

The test fails. You should modify the constructor to fulfill the expected conditions:

```
class AggregateFeed(object):
    """Aggregates several feeds"""

    def __init__(self):
        self.entries = Set()
        self.feed_factory = FeedEntry
```

The test now succeeds. The next step is refactoring `add()`:

```
def add(self, entry):
     self.entries.add(entry)
```

The tests still succeed, so the refactoring worked.

Test: Defining FeedEntry.from_parsed_feed

Now it is time to verify the `FeedEntry` factory's operation. The required feed objects already exist within the tests, and you'll reuse them here.

```
def test_feed_entry_from_parsed_feed():
    """Factory method to create a new feed entry from a parsed feed"""
    feed_entry = FeedEntry.from_parsed_feed(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_items[0]['date'], feed_entry.date)
    assert_equals(xkcd_items[0]['title'], feed_entry.title)
    assert_equals(xkcd_feed['feed']['title'], feed_entry.feed_title)
```

The test runs and fails. The method `from_parsed_feed()` is defined as follows:

```
@classmethod
def from_parsed_feed(cls, feed, entry):
    """Factory method producing a new object from an existing feed."""
    feed_entry = FeedEntry()
    feed_entry.date = entry['date']
    feed_entry.feed_title = feed['feed']['title']
    feed_entry.title = entry['title']
    return feed_entry
```

Test: Defining feed_entry_listing

At this point, `_pending_test_aggregate_item_listing()` jumps out from the list of pending tests. It pertains to `FeedEntry`, and it looks like `FeedEntry` has all the information needed.

```
def test_feed_entry_listing():
    """Should produce a correctly formatted listing from a feed entry"""
    entry = FeedEntry.from_parsed_feed(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_listings[0], entry.listing())
```

The test fails. The new method, `FeedEntry.listing()`, is defined as follows:

```
def listing(self):
    return "%s: %s: %s" % (self.date, self.feed_title, self.title)
```

The test passes, so the example is one step closer to completion.

Test: Defining feeds_from_urls

At this point, there are a few tests left. URLs must be converted into feeds, feed entries must be converted into listings, and all of the new machinery must be hooked into the `main()` method. At this point, we'll try to finish off the `AggregateFeed` by focusing on the conversion of URLs to feeds.

The test is `test_get_feeds_from_urls()`. URLs are converted to feeds via `feedparser.parse()`. This can be viewed as a factory method. The dependency is initialized in a manner analogous to `feed_factory()`.

```
def test_get_feeds_from_urls():
    """Should get a feed for every URL"""
    urls = [Mock(), Mock()]
    feeds = [Mock(), Mock()]
    subject = AggregateFeed()
    subject.feedparser = Mock()
    subject.feedparser.expects(once()).parse(same(urls[0])).will(
        return_value(feeds[0]))
    subject.feedparser.expects(once()).parse(same(urls[1])).will(
        return_value(feeds[1]))
    returned_feeds = subject.feeds_from_urls(urls)
    assert_equals(feeds, returned_feeds)
    subject.feedparser.verify()
```

The test fails. The definition fulfilling the test is as follows:

```
def feeds_from_urls(self, urls):
    """Get feeds from URLs"""
    return [self.feedparser.parse(url) for url in urls]
```

The test succeeds.

Test: AggregateFeed Initializes the FeedParser Factory

The method `feeds_from_urls()` depends on the `feedparser` property being initialized, so a test must ensure this:

```
def test_aggregate_feed_initializes_feed_parser():
    """Ensure AggregateFeed initializes dependency on feedparser"""
    assert_equals(feedparser AggregateFeed().feedparser)
```

The test fails. The initialization method is updated:

```
def __init__(self):
    self.entries = Set()
    self.feed_factory = FeedEntry
    self.feedparser = feedparser
```

The test now succeeds.

Test: Defining from_urls

At this point, you should be asking yourself the following question: where does the list of feeds get aggregated? Any time you have a question like this, it suggests that you need to write a new test to answer the question. This test should check that the method gets `feeds_from_urls()` and that it combines those feeds. The test is as follows:

```
def test_from_urls():
    """Should get feeds from URLS and combine them"""
    urls = Mock()
    aggregate_feed = Mock()
    feeds = Mock()
    aggregate_feed.expects(once()).feeds_from_urls(same(urls)).\
        will(return_value(feeds))
    aggregate_feed.expects(once()).combine_feeds(same(aggregate_feed),
        same(feeds))
    AggregateFeed().from_urls(aggregate_feed, urls)
    aggregate_feed.verify()
```

The test fails, so you write the new method:

```
def from_urls(self, feed_aggregator, urls):
    """Produce aggregated feeds from URLs"""
    feeds = feed_aggregator.feeds_from_urls(urls)
    feed_aggregator.combine_feeds(feed_aggregator, feeds)
```

The test succeeds.

Refactoring: Reimplementing from_urls

Is there any functionality still unimplemented in `AggregateFeed`? For the moment, it doesn't appear so. However, I'm not comfortable with the code as it stands—it seems overly complicated. The discomfort comes from the interactions between `from_urls()`, `feeds_from_urls()`, and `combine_feeds()`. The data flow exhibits a Y shape, as shown in Figure 7-6.

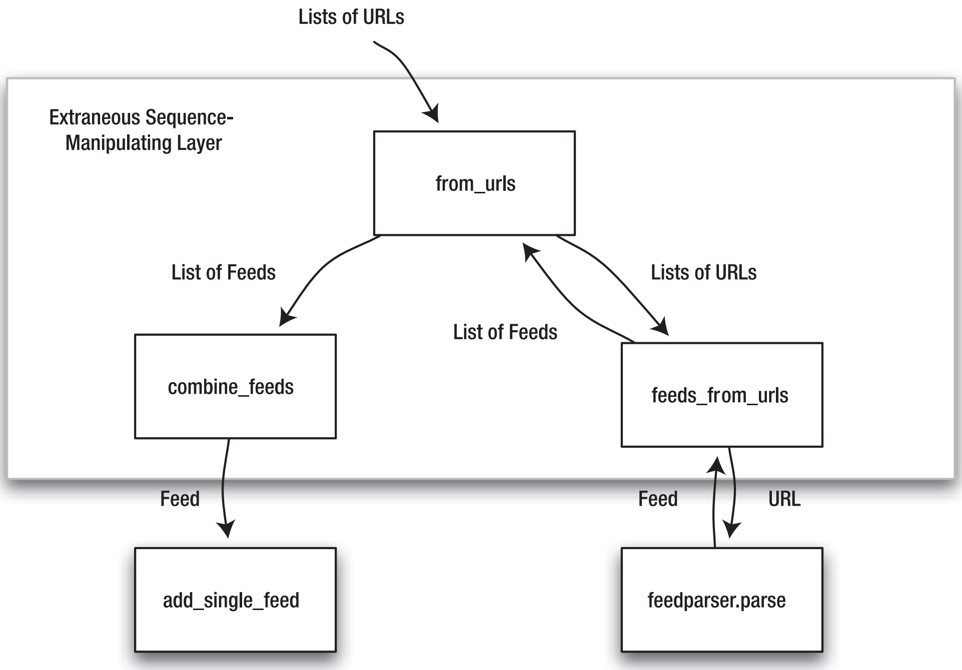


Figure 7-6. *Questionably complicated data flow*

A collection of URLs is passed down one leg, it is mapped to feeds, a collection of feeds is returned, and then the collection is fed down the other leg where another mapping is performed. It results in a layer of collection manipulations. This data flow pattern can often be transformed to a sequential set of mappings with only one iteration, as shown in Figure 7-7.

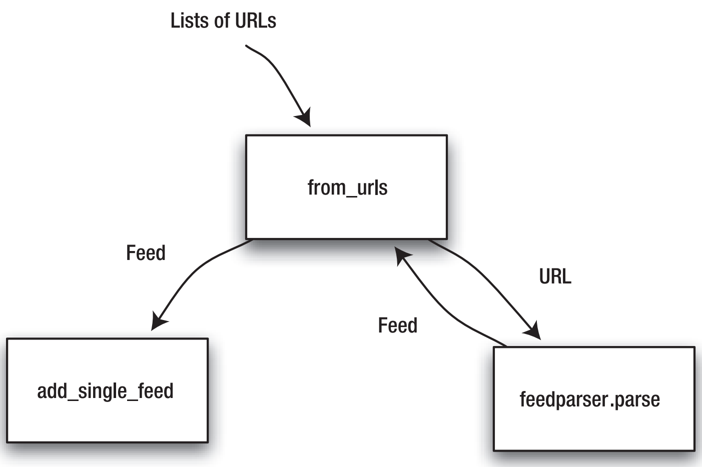


Figure 7-7. *Simplified data flow*

This rewritten test reflects the change in data flow:

```
def test_from_urls():
    """Should get feeds from URLs and combine them"""
    urls = [Mock(), Mock()]
    feeds = [Mock(), Mock()]
    subject = AggregateFeed()
    aggregate_feed = Mock()
    subject.feed_factory = Mock()
    #
    subject.feed_factory.expects(once()).parse(same(urls[0])).will(
        return_value(feeds[0]))
    aggregate_feed.expects(once()).add_single_feed(aggregate_feed, feeds[0])
    #
    subject.feed_factory.expects(once()).parse(same(urls[1])).will(
        return_value(feeds[1]))
    aggregate_feed.expects(once()).add_single_feed(aggregate_feed, feeds[1])
    #
    subject.from_url(aggregate_feed, urls)
    subject.feed_factory.verify()
    aggregate_feed.verify()
```

The test fails spectacularly. The new definition for `from_urls()` is as follows:

```
def from_urls(self, aggregate_feed, urls):
    """Produce aggregated feeds from URLs"""
    for x in urls:
        self.add_single_feed(aggregate_feed, self.feed_factory.parse(x))
```

The test passes. This definition of `from_url()` bypasses `feeds_from_urls()` and `combine_feeds()`. Those two methods and the corresponding tests can be removed. The excised code can always be retrieved from the source repository if it turns out to be needed later. (You have been checking in the code regularly, haven't you?)

One of the primary benefits of using mock objects is the style produced in the preceding example. In this style, mappings are composed and the composition action itself is tested. Without mock objects, writing for testability forces the code into a more expansive style where mappings are performed on sequences and the resulting sequences are examined. An additional layer then coordinates those sequencing operations.

Refactoring: Condensing Some Tests

What remains to be done? The tests `test_aggregate_feed_creates_factory()` and `test_aggregate_feed_initializes_feed_parser()` both verify that dependencies are initialized correctly, so they can be combined. The combined test is as follows:

```
def test_aggregate_feed_dependency_initialization():
    """"Should correctly initialize dependencies"""
    assert_equals(FeedEntry, AggregateFeed().feed_factory)
    assert_equals(feedparser, AggregateFeed().feedparser)
```

The test passes. The other two tests are removed, all tests still pass, and AggregateFeed is complete. The next set of tests examine printing.

Test: Formatting Feed Entry Listings

The printing tests should verify that individual feed listings are combined correctly, and that an unsorted feed produces sorted listings. Most of the test data for these tests already exists, so it is reused. Checking sorting on one key only requires two data points, so only two data points are used. This data is not in sorted order. Printing is a distinct category of functionality, so the printing methods will be put in a separate class.

```
from rsreader.application import AggregateFeed, FeedEntry, FeedWriter, RSReader
...
def test_aggregate_feed_listing_should_be_sorted():
    """Should produce a sorted listing of feed entries."""
    unsorted = [FeedEntry.from_parsed_feed(xkcd_feed, xkcd_items[1]),
                FeedEntry.from_parsed_feed(xkcd_feed, xkcd_items[0])]
    aggregate_feed = AggregateFeed()
    aggregate_feed.entries = unsorted
    aggregate_listing = FeedWriter().entry_listings(aggregate_feed)
    assert_equals(xkcd_output, aggregate_listing)
```

The test fails with an error, which reports that FeedWriter doesn't exist.

```
class FeedWriter(object):
    """Prints an aggregate feed"""

    def entry_listings(self, aggregate_feed):
        """Produce a sorted listing of an aggregate feed"""
        return None
```

With FeedWriter and entry_listings() defined, the test fails with an equality mismatch. The result is now faked:

```
    def entry_listings(self, aggregate_feed):
        """Produce a sorted listing of an aggregate feed"""
        entries = aggregate_feed.entries
        return os.linesep.join([entries[1].listing(),
            entries[0].listing()])
```

The test passes. The question is now how to sort the entries by date. The feed entries contain the date as a printable string, so these can't be compared directly. Fortunately, FeedParser converts them into a comparable form.

The relevant field is date_parsed. You put this field into the test data, and then you modify the FeedEntry conversion routines. The definition of xkcd_items in shared_data.py becomes the following:

```
xkcd_items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
               'date_parsed': mktime(2007, 12, 5, 5, 0, 0, 2, None),
              'title': "Python"},
```

```
        {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
         'date_parsed': mktime(2007, 12, 3, 5, 0, 0, 2, None),
         'title': "Far Away"}]
```

The tests all pass. Now `test_feed_entry_from_parsed_feed()` is modified:

```
def test_feed_entry_from_parsed_feed():
    """Factory method to create a new feed entry from a parsed feed"""
    feed_entry = FeedEntry.from_parsed_feed(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_items[0]['date'], feed_entry.date)
    assert_equals(xkcd_items[0]['date_parsed'], feed_entry.date_parsed)
    assert_equals(xkcd_items[0]['title'], feed_entry.title)
    assert_equals(xkcd_feed['feed']['title'], feed_entry.feed_title)
```

This test fails, so you rewrite `FeedEntry.from_parsed_feed()` as follows:

```
@classmethod
def from_parsed_feed(cls, feed, entry):
    """"Factory method producing a new feed entry from a feedparser entry"""
    feed_entry = FeedEntry()
    feed_entry.date = entry['date']
    feed_entry.date_parsed = entry['date_parsed']
    feed_entry.feed_title = feed['feed']['title']
    feed_entry.title = entry['title']
     return feed_entry
```

The test suite passes. You can now modify the method `entry_listings()` to support sorting:

```
def entry_listings(self, aggregate_feed):
    """Produce a sorted listing of an aggregate feed"""
    sorted_entries = sorted(aggregate_feed.entries,
                            key=lambda x: x.date_parsed,
                            reverse=True)
    return os.linesep.join([x.listing() for x in sorted_entries])
```

The test suite still passes. Now the printing behavior must be verified.

Test: Defining print_entry_listings

`print_entry_listings()` takes a listing from `FeedWriter` and prints it to `stdout`. `stdout` will be contained in an instance variable. The test needs to mock out both calls to `FeedWriter` to capture `stdout`, and it needs to mock out a call to `FeedWriter.entry_listings()`. `stdout` will be contained in an instance variable.

```
def test_print_entry_listings():
    """Verify that a listing was printed"""
    subject = FeedWriter()
    (feed_writer, aggregate_feed, listings) = (Mock(), Mock(), Mock())
    subject.stdout = Mock()
```

```
    feed_writer.expects(once()).entry_listings(same(aggregate_feed)).\
        will(return_value(listings))
    subject.stdout.expects(once()).write(same(listings))
    subject.stdout.expects(once()).write(eq(os.linesep))
    subject.print_entry_listings(feed_writer, aggregate_feed)
    feed_writer.verify()
    subject.stdout.verify()
```

The test fails. The printing code is implemented:

```
def print_entry_listings(self, feed_writer, aggregate_feed):
    """Print listing"""
    self.stdout.write(feed_writer.entry_listings(aggregate_feed))
    self.stdout.write(os.linesep)
```

The test succeeds. `FeedListing.stdout` needs to be initialized, and the next test ensures this.

Test: FeedWriter Initializes the stdout Attribute

The `stdout` attribute must be initialized when the `FeedWriter` creates itself.

```
def test_feed_writer_intializes_stdout():
    """Ensure that feed writer initializes stdout from sys.stdout"""
    assert_equals(sys.stdout, FeedWriter().stdout)
```

The test fails. The initialization code is written as follows:

```
class FeedWriter(object):
    """Prints an aggregate feed"""

    def __init__(self):
        self.stdout = sys.stdout
```

The test succeeds. The final printing test ensures that nothing is printed if there are no entries.

Test: Empty AggregateFeeds Generate No Output

When the feeds have no items, then the program should produce no output. This test ensures that.

```
def test_feed_writer_prints_nothing_with_an_empty_feed(self):
    """Empty aggregate feed should print nothing"""
    subject = FeedWriter()
    (feed_writer, aggregate_feed) = (Mock(), Mock())
    subject.stdout = Mock()
    aggregate_feed.expects(once()).is_empty().will(return_value(True))
    subject.print_entry_listing(feed_writer, aggregate_feed)
    aggregate_feed.verify()
    subject.stdout.verify()
```

The test fails. The method is updated:

```
def print_entry_listings(self, feed_writer, aggregate_feed):
    """Print listing"""
    if not aggregate_feed.is_empty():
        self.stdout.write(feed_writer.entry_listings(aggregate_feed))
        self.stdout.write(os.linesep)
```

The test succeeds, but `test_print_entry_listing()` fails because it doesn't mock out the call to `aggregate_feed.is_empty()`. The broken test now reads as follows:

```
def test_print_entry_listing():
    """Verify that a listing was printed"""
    subject = FeedWriter()
    (feed_writer, aggregate_feed, listings) = (Mock(), Mock(), Mock())
    subject.stdout = Mock()
    aggregate_feed.expects(once()).is_empty().will(
        return_value(False))
    feed_writer.expects(once()).entry_listings(same(aggregate_feed)).\
        will(return_value(listings))
    subject.stdout.expects(once()).write(same(listings))
    subject.stdout.expects(once()).write(eq(os.linesep))
    subject.print_listings(feed_writer, aggregate_feed)
    feed_writer.verify()
    subject.stdout.verify()
```

The test now succeeds. The new method `AggregateFeed.is_empty()` must be tested.

Test: Defining is_empty

The preceding test used the `is_empty()` method. This is a good time to create it.

```
def test_is_empty():
    """Unsure is empty works"""
    aggregate_feed = AggregateFeed()
    assert aggregate_feed.is_empty()
    aggregate_feed.add("foo")
    assert not aggregate_feed.is_empty()
```

The test fails, so you define `is_empty()` as follows:

```
def is_empty(self):
    """True if set is empty, and False otherwise"""
    return not self.entries
```

The test now succeeds, and printing is complete.

Test: Defining new_main

The `new_main()` method must be tied into the existing program. It should get `AggregateFeed` objects from a set of URLs, and then it should print them.

When argv is passed into the method, it contains extra information that must be removed. The first argument is the program name, and the subsequent arguments are the URLs to be read. This test ensures that the unneeded leading argument is stripped off.

```
def test_new_main():
    """"Main should create a feed and print results"""
    args = ["unused_program_name", "x1"]
    reader = RSReader()
    reader.aggregate_feed = Mock()
    reader.feed_writer = Mock()
    reader.aggregate_feed.expects(once()).from_urls(same(reader.aggregate_feed),
         eq(["x1"]))
    reader.feed_writer.expects(once()).print_listings(same(reader.aggregate_feed))
    reader.new_main(args)
    reader.aggregate_feed.verify()
    reader.feed_writer.verify()
```

The test fails. The new_main() function is as follows:

```
def new_main(self, argv):
    self.aggregate_feed.from_urls(self.aggregate_feed, argv[1:])
    self.feed_writer.print_listings(self.aggregate_feed)
```

The test succeeds. Now you have to check that aggregate_feed and feed_writer have been initialized.

Test: The Application Initializes Dependencies

The application needs to initialize its dependencies, and this test ensures that it does so:

```
def test_rsreader_initializes_dependencies():
    """RSReader should initialize dependencies"""
    reader = RSReader()
    assert isinstance(reader.aggregate_feed, AggregateFeed)
    assert isinstance(reader.feed_writer, FeedWriter)
```

The test fails. The __init__ method is implemented as follows:

```
class RSReader(object):
    """The Application"""

    def __init__(self):
        self.aggregate_feed = AggregateFeed()
```

And the test gets a bit further. The method is expanded:

```
    def __init__(self):
        self.aggregate_feed = AggregateFeed()
        self.feed_writer = FeedWriter()
```

And the test succeeds. At this point, the new application code is complete, but the script is still executing the old main() method.

Refactoring: Making new_main the New main[2]

You're now ready to make the changes active by replacing `main()` with `new_main()`. This happens with the tests, too—you rename the `test_new_main()` method to `test_main()`:

```
def test_main():
    """"Main should create a feed and print results"""
    args = ["unused_program_name", "x1"]
    reader = RSReader()
    reader.aggregate_feed = Mock()
    reader.feed_writer = Mock()
    reader.aggregate_feed.expects(once()).from_urls(same(reader.aggregate_feed),
       eq(["x1"]))
    reader.feed_writer.expects(once()).print_listings(same(reader.aggregate_feed))
    reader.main(args)
    reader.aggregate_feed.verify()
    reader.feed_writer.verify()
```

The test fails. You fix this by renaming `new_main()` to `main()`:

```
def main(self, argv):
    self.aggregate_feed.from_urls(self.aggregate_feed, argv[1:])
    self.feed_writer.print_listings(self.aggregate_feed)
```

The new tests all succeed, as do the acceptance tests. The old tests fail, but that's OK since you're about to remove them. You should take note of the failing tests, and then remove the extraneous code that these methods test. If more tests fail, then you may have removed too much. If the acceptance tests still pass, then you should remove the failing tests for old functionality. At this point, the program is complete.

A Simple PyMock Example

A simple test shows how PyMock is used. The example will calculate a triangle's perimeter:

```
def test_perimeter():
    assert_equals(4, perimeter(triangle))
```

PyMock must be initialized before you can use it in a function or method. This is done with the `use_pymock` decorator:

```
@use_pymock
def test_perimeter():
    assert_equals(4, perimeter(triangle))
```

Here, PyMock is used to imitate `triangle`:

2. War is the new peace. Politics is the new gossip. Pink is the new black. Whales are the new sushi.

```
@use_pymock
def test_perimeter():
    triangle = mock()
    assert_equals(4, perimeter(triangle))
```

PyMock uses a record-replay mechanism. Expectations are set either by performing calls exactly as they are expected to be replayed, or by describing them with a DSL similar to pMock. I'll demonstrate the former here:

```
@use_pymock
def test_perimeter():
    triangle = mock()
    triangle.side[0]; returns(1); once()
    triangle.side[1]; returns(3); once()
    assert_equals(4, perimeter(triangle))
```

The direct recording makes it easy to record multistep sequences. The actions here have two steps: the property `side` is retrieved, and then `side.__getitem__()` is called.[3] PyMock uses `__eq__()` to compare most arguments, with mocks being the sole exception; they are compared by identity. The return values and counts are specified using additional functions.

After expectations have been set, the mock is switched from record mode to replay mode:

```
@use_pymock
def test_perimeter():
    triangle = mock()
    triangle.side[0]; returns(1); once()
    triangle.side[1]; returns(3); once()
    replay()
    assert_equals(4, perimeter(triangle))
```

Here's the simplest method fulfilling the test:

```
def perimeter(triangle):
    return 4
```

The test passes, but it shouldn't. You want to verify that the `triangle` object was used, and you do this by calling `verify()`. This function checks all recorded expectations. This contrasts with pMock, in which the `verify()` method must be called for each mock that you want to check.

```
def test_perimeter():
    triangle = Mock()
    triangle.side[0]; returns(1); once()
    triangle.side[1]; returns(3); once()
    replay()
    assert_equals(4, perimeter(triangle))
    verify()
```

3. Every call to a mock returns a new mock unless you specify something else using `returns()` or `raises()`.

Now the test fails. The following definition satisfies the test:

```
def perimeter(triangle):
   return triangle.side[0] + triangle.side[1]
```

It is worth noting that the call count is optional. If it is not specified, then `once()` is assumed, so the following code is equivalent to the preceding test:

```
def test_perimeter():
    triangle = Mock()
    triangle.side[0]; returns(1)
    triangle.side[1]; returns(3)
    replay()
    assert_equals(4, perimeter(triangle))
    verify()
```

Monkeypatching

PyMock directly supports monkeypatching existing classes, attributes, and properties. This is one of the primary distinctions between PyMock and pMock. This allows your code to temporarily override an existing package or attribute, without having to explicitly inject dependencies. Monkeypatching is done with the `override()` function, as shown here:

```
@use_pymock
def test_dirpaths(self):
    root = 'path'
    listed_directories = ['one', 'two']
    expected_paths = [os.path.join(root, 'one'), os.path.join(root, 'two')]
    override(os, 'listdir'); os.listdir(root); returns(listed_directories)
    replay()
    assert_equals(expected_paths, dirpaths(root))
    verify()
```

The definition fulfilling the test is

```
def dirpatch(root):
    return [os.path.join(root, x) for x in os.listdir(root)]
```

Saying the Same Thing Differently

PyMock supports a second syntax to define expectations. It is closer to pMock's, and it suffers similar limitations. It is restricted to simple function calls, but it is more concise when monkeypatching. The previous test becomes the following:

```
@use_pymock
def test_dirpaths(self):
    root = 'path'
    listed_directories = ['one', 'two']
    expected_paths = [os.path.join(root, 'one'), os.path.join(root, 'two')]
```

```
    override(os, 'listdir').expects(root).returns(listed_directories)
    replay()
    assert_equals(expected_paths, dirpaths(root))
    verify()
```

The override expression now completely specifies the expected call. The arguments are specified with `expects()`, and the return values are specified with `returns()`.

Calls on mock objects are specified with the `method()` function:

```
def test_perimeter():
    triangle = Mock()
    method(triangle, 'side').expects(0).returns(1)
    method(triangle, 'side').expects(1).returns(3)
    replay()
    assert_equals(4, perimeter(triangle))
    verify()
```

This test specifies this function:

```
def perimeter(triangle):
   return triangle.side(0) + triangle.side(1)
```

Implementing with PyMock

The implementation with PyMock largely mirrors the pMock implementation, but there are significant differences. Monkeypatching eliminates the necessity of introducing a second variable to hold the mock. Instead, the mocked method is attached directly to an instance using the `override()` function. This leads to a simpler call structure.

Monkeypatching also allows tests to mock external modules in place. In cases where the dependencies are never expected to change, this leads to a simpler application. Modules are referenced directly rather than being referenced through variables.[4]

The tests will start with `from_urls()`. This method wasn't in the pending tests originally brainstormed. It was discovered along the way, but it is a good place to start. It obviates the need for both `get_feeds_from_urls()` and `combine_feeds()`, so those pending tests are discarded. `test_feed_entry_listing()` duplicates `test_feed_listing_is_sorted()`, so that will be ignored, too. There is nothing to be gained from repeating material with no new insight, so the list of pending tests is now as follows:

```
def _pending_test_new_main():
    """Hooking in new code to main"""

def _pending_test_from_urls():
    """Should retrieve feeds and add them to the aggregate"""
```

4. Some might say, "Rather than being injected as dependencies," but we're not dealing with Java here.

```
def _pending_test_get_feeds_from_urls():
    """Should get a feed for every URL"""

def _pending_test_combine_feeds():
    """Should combine feeds into a list of FeedEntries"""

def _pending_test_add_single_feed():
    """Should add a single feed to a set of feeds"""

def _pending_test_create_entry():
    """Create a feed item from a feed and a feed entry"""

def _pending_test_feed_listing_is_sorted():
    """Should sort the aggregate feed listing"""

def _pending_test_feed_entry_listing():
    """Should produce a correctly formatted listing from a feed entry"""
```

Test: from_urls and Mocking External Modules

From the very beginning, you'll make strong use of `override()`:

```
import feedparser
...
@use_pymock
def test_from_urls():
    """Should retrieve feeds and add them to the aggregate"""
    urls = [dummy(), dummy()]
    feeds = [dummy(), dummy()]
    subject = RSReader()
    #
    override(feedparser, 'parse').expects(urls[0]).returns(feeds[0])
    override(subject, 'add_single_feed').expects(feeds[0])
    #
    override(feedparser, 'parse').expects(urls[1]).returns(feeds[1])
    override(subject, 'add_single_feed').expects(feeds[1])
    #
    replay()
    subject.from_urls(urls)
    verify()0
```

The `dummy()` calls create dummy objects. They're impostors with no functionality that exist only to be used as arguments. In actuality, the objects returned from `dummy()` are full-fledged mock objects—just the same as those returned from `mock()`—but the factory method makes the intention clear to the test's readers.

The test fails to execute, but defining `from_urls()` is insufficient. The `override()` function verifies the existence of `add_single_feed()`, so it must be defined before the test can run:

```
def from_urls(self, feeds):
    """"Combine a set of parsed feeds"""

def add_single_feed(self, feed):
    """Add a single parsed feed"""
```

The test now raises a verification failure. The full definition satisfying the test is as follows:

```
def  from_urls(self, feeds):
    """"Combine a set of parsed feeds"""
    for f in feeds:
        self.add_single_feed(f)

def add_single_feed(self, feed):
    """Add a single parsed feed"""
```

The tests runs, and it succeeds.

Test: Defining add_single_feed

The next test characterizes `add_single_feed()`.The method `add_single_feed()` should call `create_entry()` once for each entry in the feed the caller passes in.

```
@use_pymock
def test_add_single_feed():
    """Should create a new entry for each entry in the feed"""
    reader = RSReader()
    entries = [dummy(), dummy()]
    feed = mock()
    feed.entries; returns(entries)
    override(reader, 'create_entry').expects(entries[0])
    override(reader, 'create_entry').expects(entries[1])
    replay()
    reader.add_single_feed(feed)
    verify()
```

The test fails to execute because `create_entry()` hasn't been defined yet. Here is a minimal definition:

```
def add_single_feed(self, feed):
    """Add a single parsed feed"""

def create_entry(self, feed, entry):
    """Add a single entry"""
```

The test now fails with a verification exception. The method is defined as follows:

```
def add_single_feed(self, feed):
    """Add a single parsed feed"""
    for x in urls:
        self.add_single_feed(feedparser.parse(x))
```

```
    def create_entry(self, feed, entry):
        """Add a single entry"""
```

The test now passes.

Refactoring: Moving Methods to a New Object

At this point in the pMock example, it was clear that these methods belonged in their own class. The explicit dependency that had to be passed in like a second `self` is missing. Monkeypatching bypasses the need to inject a dependency, but it was this very dependency that made it clear where these methods belonged.[5] The trade-off is that the resulting code looks more Pythonic and less alien.

Refactoring: Moving add_single_feed

Moving the methods to another class highlights another difference involved with gleeful monkeypatching. Methods have to be moved in twos. One method is moved, and the monkeypatched one is temporarily duplicated. I prefer to start with the last implemented method. The test becomes the following:

```
from rsreader.app import AggregateFeed, RSReader
...
def test_add_single_feed():
    """Should create a new entry for each entry in the feed"""
    subject = AggregateFeed()
    entries = [dummy(), dummy()]
    feed = mock()
    feed.entries; returns(entries)
    override(subject, 'create_entry').expects(feed, entries[0])
    override(subject, 'create_entry').expects(feed, entries[1])
    replay()
    subject.add_single_feed(feed)
```

The test fails because `AggregateFeed` hasn't been created. The class is created, and then the test fails because the methods haven't been defined. At this point, I'll move over the methods `add_single_feed()` and `create_entry()`. The first is a complete method, while the second is a stub. One of the implicit goals in TDD is never breaking more than one test at a time. If other tests depended on `RSReader.create_entry()`, then I would leave a copy behind, and I would only remove it after altering those tests.

```
class RSReader(object):
    """The Application"""

    def from_urls(self, urls):
        """Transform URLS into feeds"""
```

5. You could restrict your usage of `override()` to standard library modules such as `os` or `sys`, but that doesn't seem to work in practice—it's too tempting a tool.

```
        for x in urls:
            self.add_single_feed(feedparser.parse(x))

    def add_single_feed(self, feed):
        """Add a single parsed feed"""


class AggregateFeed(object):
    """Several parsed feeds combined"""

    def add_single_feed(self, feed):
        """Add a single parsed feed"""
        for e in feed.entries:
            self.create_entry(feed, e)

    def create_entry(self, feed, entry):
        """Add a single entry"""
```

The test succeeds.

Refactoring: Moving from_urls()

The method `from_urls()` belongs in the class `AggregateFeed`, so you modify `test_from_urls()` to expect this change:

```
@use_pymock
def test_from_urls():
    """Should retrieve feeds and add them to the aggregate"""
    urls = [dummy(), dummy()]
    feeds = [dummy(), dummy()]
    subject = AggregateFeed()
    #
    override(feedparser, 'parse').expects(urls[0]).returns(feeds[0])
    override(subject, 'add_single_feed').expects(feeds[0])
    #
    override(feedparser, 'parse').expects(urls[1]).returns(feeds[1])
    override(subject, 'add_single_feed').expects(feeds[1])
    #
    replay()
    subject.from_urls(urls)
    verify()
```

The test fails, so you move `from_url()` from RSReader to `AggregateFeed`. There are no more tests depending on the stub method `add_single_feed()`, so you remove it:

```
class RSReader(object):
    """The Application"""
```

```
class AggregateFeed(object):
    """Several parsed feeds combined"""

    def from_urls(self, urls):
        """Transform URLS into feeds"""
        for x in urls:
            self.add_single_feed(feedparser.parse(x))

    def add_single_feed(self, feed):
        """Add a single parsed feed"""
        for e in feed.entries:
            self.create_entry(feed, e)

    def create_entry(self, feed, entry):
        """Add a single entry"""
```

The test succeeds.

Test: create_entry() and Mocking Class Constructors

With pMock, it was necessary to use a factory to introduce the relationship between `AggregateFeed.create_entry()` and the `FeedEntry` objects it produces. With PyMock, the constructor for `FeedEntry` is mocked out directly.

I can argue that using a factory makes for a better design by explicitly capturing the dependency. I can also argue that it is overkill for this application. When the class is referenced in more than one place, or when it comes time to introduce a second kind of element, then a factory may be the better choice.

```
import rsreader.application
...
@use_pymock
def test_create_entry():
    """Should create an entry and add it to the collection"""
    subject = AggregateFeed()
    (feed, entry) = (dummy(), dummy())
    new_entry = dummy()
    override(rsreader.application, 'FeedEntry')\
        .expects(feed, entry)\
        .returns(new_entry)
    override(subject, 'add').expects(new_entry)
    replay()
    subject.create_entry(feed, entry)
    verify()
```

The test fails to run because `FeedEntry` is not defined. The definition is as follows:

```
class FeedEntry(object):
    """Combines elements of a feed and a feed entry.
    Allows multiple feeds to be aggregated without losing feed specific
    information."""
```

The test progresses a bit further before failing with an error. It now complains that `add()` isn't defined, so you stub it out:

```
    def create_entry(self, feed, entry):
        """Add a single entry"""

    def add(self, entry):
        """Add an entry"""
```

The test now fails with a verification error. You complete `create_entry()`:

```
    def create_entry(self, feed, entry):
        """Add a single entry"""
        self.add(FeedEntry(feed, entry))

    def add(self, entry):
        """Add an entry"""
```

With this definition, the test succeeds.

Tests: Defining add and AggregateFeed.__init__

These two tests are almost exactly the same as with pMock, with only the slightest differences between the `add()` implementations. I'll just summarize them here:

```
from sets import Set

from nose.tools import *
from pymock import mock, override, replay, returns, verify, use_pymock

from rsreader.application import AggregateFeed, FeedEntry, RSReader

...

    @use_pymock
    def test_add():
        """Add an a feed entry to the aggregate"""
        entry = mock()
        subject = AggregateFeed()
        subject.add(entry)
        assert_equals(Set([entry]), subject.entries)

    def test_entries_is_always_defined():
        """The entries set should always be defined"""
        assert_equals(Set(), AggregateFeed().entries)
```

The code satisfying these tests is as follows:

```
from sets import Set
...

    def __init__(self):
        self.entries = Set()
...

    def add(self, entry):
        """Add to the set of entries"""
        self.entries.add(entry)
```

With these definitions, the tests once again successfully run to completion.

Test: Defining FeedEntry.__init__

`FeedEntry` construction has been mocked out, but the `FeedEntry` constructor doesn't exist yet. With pMock, there is no way to mock `__init__()`, so the constructor was a class method. With PyMock, you can mock `__init__()` directly:

```
def test_feed_entry_constructor():
    """Verify settings extracted from feed and entry"""
    subject = FeedEntry(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_items[0]['date'], subject.date)
    assert_equals(xkcd_items[0]['title'], subject.title)
    assert_equals(xkcd_feed['feed']['title'], subject.feed_title)
```

The test fails, so you redefine `__init__` as follows:

```
def __init__(self, feed, entry):
    self.date = entry['date']
    self.feed_title = feed['feed']['title']
    self.title = entry['title']
```

The test now succeeds.

Test: Defining listing

The next test ensures that feed entry listings are correctly formatted:

```
def test_feed_entry_listing():
    """Should produce a correctly formatted listing form a feed item"""
    subject = FeedEntry(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_listings[0], subject.listing())
```

The test fails. The code producing the correct listing is as follows:

```
def listing(self):
    return "%s: %s: %s" % (self.date, self.feed_title, self.title)
```

The test succeeds.

Test: entry_listings Should Be Sorted

The next set of tests implement printing. The first test, and the ripple of changes resulting from it, are nearly the same as in the pMock example. Those tests are as follows:

```
from rsreader.application import AggregateFeed, FeedEntry, FeedWriter, RSReader

...

def test_feed_entry_constructor():
    """Verify settings extracted from feed and entry"""
    subject = FeedEntry(xkcd_feed, xkcd_items[0])
    assert_equals(xkcd_items[0]['date'], subject.date)
    assert_equals(xkcd_items[0]['date_parsed'], subject.date_parsed)
    assert_equals(xkcd_items[0]['title'], subject.title)
    assert_equals(xkcd_feed['feed']['title'], subject.feed_title)
...

def test_aggregate_feed_listing_should_be_sorted():
    """Should produce a sorted listing of feed entries."""
    unsorted_entries = [FeedEntry(xkcd_feed, xkcd_items[1]),
                        FeedEntry(xkcd_feed, xkcd_items[0])]
    aggregate_feed = AggregateFeed()
    aggregate_feed.entries = unsorted_entries
    assert_equals(xkcd_output, FeedWriter().entry_listings(aggregate_feed))
```

The following change is made to the test data:

```
xkcd_items = [{'date': "Wed, 05 Dec 2007 05:00:00 -0000",
           'date_parsed': mktime(2007, 12, 5, 5, 0, 0, 2, None),
          'title': "Python"},
         {'date': "Mon, 03 Dec 2007 05:00:00 -0000",
          'date_parsed': mktime(2007, 12, 3, 5, 0, 0, 2, None),
          'title': "Far Away"}]
```

The changes and additions, made hand-in-hand with the preceding tests, are the following:

```
class FeedEntry(object):
    """Combines elements of a feed and a feed entry.
    Allows multiple feeds to be aggregated without losing feed specific
    information."""

    def __init__(self, feed, entry):
        self.date = entry['date']
        self.date_parsed = entry['date_parsed']
        self.feed_title = feed['feed']['title']
        self.title = entry['title']
```

```
    def listing(self):
        return "%s: %s: %s" % (self.date, self.feed_title, self.title)


class FeedWriter(object):
    """Prints an aggregate feed"""

    def entry_listings(self, aggregate_feed):
        """Produce a sorted listing of an aggregate feed"""
        sorted_entries = sorted(aggregate_feed.entries,
                                key=lambda x: x.date_parsed,
                                reverse=True)
        return os.linesep.join([x.listing() for x in sorted_entries])
```

Once these changes have been made, the tests succeed.

Test: Defining print_entry_listings

With pMock, printing was performed through a local copy of `sys.stdout`, which was held in `FeedWriter.stdout`. This instance variable was replaced with a mock. PyMock can easily mock the method in situ.

```
def test_print_agg_feed_listing_is_printed():
    """Should print listing of feed entries"""
    unsorted_entries = [FeedEntry(xkcd_feed, xkcd_items[1]),
                        FeedEntry(xkcd_feed, xkcd_items[0])]
    aggregate_feed = AggregateFeed()
    aggregate_feed.entries = unsorted_entries
    override(sys.stdout, 'write').expects(xkcd_output + os.linesep)
    replay()
    FeedWriter().print_entry_listings(aggregate_feed)
    verify()
```

The test fails. It reports `This object can't be modified`. There are some objects that Python can't monkeypatch. As a result, the object itself has to be mocked:

```
@use_pymock
def test_print_entry_listing():
    """Should print listing of feed entries"""
    unsorted_entries = [FeedEntry(xkcd_feed, xkcd_items[1]),
                        FeedEntry(xkcd_feed, xkcd_items[0])]
    aggregate_feed = AggregateFeed()
    aggregate_feed.entries = unsorted_entries
    override(sys, 'stdout')
    method(sys.stdout, 'write').expects(xkcd_output + os.linesep)
    replay()
    FeedWriter().print_entry_listings(aggregate_feed)
    verify()
```

The test fails because the method isn't defined yet. The method declaration is as follows:

```
def print_entry_listings(self, aggregate_feed):
    """Print an entry_listing to sys.stdout"""
```

The test now fails in the desired way. The full definition for the method is as follows:

```
def print_entry_listings(self, aggregate_feed):
    """Print an entry_listing to sys.stdout"""
    sys.stdout.write(self.entry_listings(aggregate_feed) + os.linesep)
```

The test succeeds, but the method isn't complete.

Test: print_entry_listings Should Do Nothing with Empty Feeds

As defined, the method will print `os.linesep` when the `AggregateFeed` is empty. It should print nothing. This test expresses that requirement:

```
def test_print_entry_listing_does_nothing_with_an_empty_aggregate():
    """Ensure that nothing is printed with an empty aggregate"""
    empty_aggregate_feed = AggregateFeed()
    override(sys, 'stdout')
    replay()
    FeedWriter().print_entry_listings(empty_aggregate_feed)
    verify()
```

This test makes use of negative assertions. No actions are defined on the mock in `sys.stdout`, ensuring that an error will arise if any are performed on it. As currently defined, `print_entry_listing()` always writes `os.linesep` to `sys.stdout`, so the test fails. The subject is redefined as follows:

```
def print_entry_listings(self, aggregate_feed):
    """Print an entry_listing to sys.stdout"""
    if not aggregate_feed.is_empty():
        entry_listings = self.entry_listings(aggregate_feed)
        sys.stdout.write(entry_listings + os.linesep)
```

Both `print_entry_listings()` tests now fail because the method `AggregateFeed.is_empty()` doesn't exist, so you define the method as follows:

```
class AggregateFeed(object):
    """Several parsed feeds combined"""

    def __init__(self):
        """Define factory"""
        self.entries = Set()

    def is_empty(self):
        """True if empty, False otherwise"""
        return not self.entries
```

The tests now pass, but there is a problem with them.

Test: is_empty and the Unproven Test

The `is_empty()` method is only tested indirectly. If it is ever broken, then the malfunction will show itself indirectly through the previous two tests. Failures should be exhibited directly, so it needs to be tested directly:

```
def test_is_empty():
    """Ensure that is_empty reports emptiness as expected"""
    empty_aggregate_feed = AggregateFeed()
    non_empty_aggregate_feed = AggregateFeed()
    non_empty_aggregate_feed.add("foo")
    assert empty_aggregate_feed.is_empty() is True
    assert non_empty_aggregate_feed.is_empty() is False
```

The test succeeds, which is a problem. It is unproved that this test fails. There are two approaches to fixing this. One is to go back several steps and mock out `is_empty()`, and slowly build up the tests. The other way is to break `is_empty()`, and verify that the test breaks. You do this by changing the return value to `None`, running the test, verifying that it failed, changing it to `True`, running the test, verifying that it failed again, and then putting back the real implementation.

This second approach is sometimes required when using automatic refactorings, particularly method or class extractions. The newly created methods and classes don't have any direct tests, so these tests must be created de novo. You'll often have to verify that the tests fail by breaking the newly refactored code and then restoring it.

Test: new_main, Hooking It All Together

The test for `new_main()` is precisely analogous to the pMock example. It verifies that the first argument is stripped off, and that the appropriate calls are made to the `AggregateFeed` and `FeedWriter` objects.

```
@use_pymock
def test_new_main():
    """Hook components together"""
    args = ["unused_program_name", "u1"]
    subject = RSReader()
    subject.aggregate_feed = mock()
    subject.feed_writer = mock()
    method(subject.aggregate_feed, 'from_urls').expects(["u1"])
    method(subject.feed_writer, 'print_entry_listings').\
         expects(subject.aggregate_feed)
    replay()
    subject.new_main(args)
    verify()
```

The test fails. The method satisfying the test is as follows:

```
def new_main(self, argv):
    """Read argument lists and coordinate aggregates"""
    self.aggregate_feed.from_urls(argv[1:])
    self.feed_writer.print_entry_listings(self.aggregate_feed)
```

The test succeeds.

Test: RSReader Initialization

One more test remains. The new `main()` method depends on the two aggregates. These dependencies must be initialized and verified. The test is as follows:

```
def test_rsreader_dependency_initialization():
    """Ensure that dependencies are correctly initialized"""
    assert isinstance(RSReader().aggregate_feed, AggregateFeed)
    assert isinstance(RSReader().feed_writer, FeedWriter)
```

The test fails. The method fulfilling the test is as follows:

```
class RSReader(object):
    """The Application"""

    def __init__(self):
        self.aggregate_feed = AggregateFeed()
        self.feed_writer = FeedWriter()
```

The test succeeds.

Finishing Up: Activating the New Functionality

The new `main()` function is complete. The entire new application has been wired together in parallel with the existing code. Now it is time activate it. The old `test_main()` is no longer needed. It is removed, and `test_new_main()` is renamed to `test_main()`. Here's the new test:

```
@use_pymock
def test_main():
    """Hook components together"""
    args = ["unused_program_name", "u1"]
    subject = RSReader()
    subject.aggregate_feed = mock()
    subject.feed_writer = mock()
    method(subject.aggregate_feed, 'from_urls').expects(["u1"])
    method(subject.feed_writer, 'print_entry_listings').\
        expects(subject.aggregate_feed)
    replay()
    subject.main(args)
    verify()
```

The test fails. The old `main()` method is removed and `new_main()` is renamed to `main()`:

```
def main(self, argv):
    """Read argument lists and coordinate aggregates"""
    self.aggregate_feed.from_urls(argv[1:])
    self.feed_writer.print_entry_listings(self.aggregate_feed)
```

With this, thePyMock application is complete.[6]

Other pMock and PyMock Features

pMock and PyMock have a number of features that haven't been covered here. Most notable are exception mocking and playback limits. PyMock also supports setter mocking and generator mocking.

Raising Exceptions with pMock

pMock uses the `raise_exception(exc)` function in the `will` clause to declare that the method raises an exception when played back:

```
def test_raising_exception():
    """Raise an exception"""
    m = Mock()
    m.expects(once()).whoops().will(raise_exception(Exception()))
    assert_raises(Exception, m.whoops)
    m.verify()
```

Raising Exceptions with PyMock

The PyMock equivalents are the `raises(exc)` method and function:

```
@use_pymock
def test_raising_exception():
    """Raise an exception"""
    m = mock()
    method(m, 'whoops').expects().raises(Exception())
    replay()
    assert_raises(Exception, m.whoops)
    verify()
```

Using the direct recording interface, this same test would be the following:

```
@use_pymock
def test_raising_exception():
    """Raise an exception"""
```

6. Go take a break. Get yourself a beer. Write someone a letter. Call your friends. You deserve it. You made it through that slog. I'm going to do the same in a few pages. I'll catch up with you.

```
    m = mock()
    m.whoops(); raises(Exception())
    replay()
    assert_raises(Exception, m.whoops)
    verify()
```

Playback Counts with pMock

pMock recognizes three different calling policies: `once()`, `at_least_once()`, and `never()`. All are used in the `expects` clause.

Playback Counts with PyMock

PyMock recognizes the following playback counts: `once()` (the default), `one_or_more()`, `zero_or_more()`, `set_count(int)`, and `at_least(int)`. Only the last two require explanation. The call `set_count()` specifies the precise number of playbacks expected, and the call `at_least()` specifies the minimum number of playbacks expected.

Mocking Attribute Setters with PyMock

PyMock mocks property getting and setting using both the raw recording style and a declarative style. Here are three different setter expressions:

```
@use_pymock
def test_setting_attributes():
    """Set attributes"""
    m = mock()
    m.f = 1
    m.f = 2; raises(Exception())
    set_attr(m, 'f', 3).once()
    replay()
    m.f = 1
    try:
        m.f = 2
    except Exception:
        pass
    m.f = 3
    verify()
```

Here are two getter expressions:

```
@use_pymock
def test_getting_attributes():
    """Get attributes"""
    m = mock()
    m.g; returns(1)
    get_attr(m, 'h').returns(2)
    replay()
```

```
    assert_equals(m.g == 1)
    assert_equals(m.h == 2)
    verify()
```

Mocking Generators with PyMock

With PyMock, generators are most clearly mocked with the declarative style:

```
@use_pymock
def testGeneratesWithRaisedTermination(self):
    m = mock()
    method(m, 'f').expects().generates(1, 2)
    replay()
    g = m.f()
    assert_equals(1, g.next())
    assert_equals(2, g.next())
    assert_raises(StopIteration, g.next)
```

You specify a terminal exception using the `ending` keyword:

```
@use_pymock
def testGeneratesWithRaisedTermination(self):
    m = mock()
    method(m, 'f').expects().generates(1, 2, ending=StopMe())
    replay()
    g = m.f()
    assert_equals(1, g.next())
    assert_equals(2, g.next())
    assert_raises(StopMe, g.next)
```

If you use the recording mode, the first example would be

```
generator(m.f(), [1, 2])
```

For the second example, the equivalent line would be

```
generator(m.f(), [1, 2], StopMe())
```

Using PyMock with unittest

PyMock defines a subclass of `unittest.TestCase` called `PyMockTestCase`. By subclassing it, all test methods are automatically configured to use PyMock. There is one caveat. `PyMockTestCase` uses `setUp()` and `tearDown()` to configure the mocking machinery and to restore monkey-patches. If you have defined your own `setUp()` or `tearDown()` methods, then they must use `super` to call the appropriate methods in the parent class.

Summary

I walked you through several TDD examples in this chapter, demonstrating how to use mock objects in excruciating detail. Throughout the chapter, example refactorings were shown and briefly discussed. Along the way, I briefly introduced the automatic refactoring tools available in Pydev.

The primary focus of the chapter was on code isolation through *impostors*. Impostors are test objects that replace application objects. Impostors, sometimes called *test doubles*, come in four different flavors. In order of increasing complication, they are *dummies*, *stubs*, *mocks*, and *fakes*.

Two mock object packages were introduced. They reflect the two different schools of thought. One focuses on restricting functionality to improve design, and the other focuses on testing ease. pMock reflects the first school of thought, and PyMock reflects the second.

pMock is modeled on Java's jMock library. It uses a *DSL* to declare expectations. PyMock is modeled on Java's EasyMock library, although it has acquired some of jMock's trappings. It uses a record-replay model. Calls are specified by performing them, and then the mock is switched into replay mode.

I compared the two packages by building the same example with each. The results were noticeably different, reflecting the capabilities and constraints of each package. This emphasizes that TDD and mock objects are design tools.

One technique introduced along the way was *monkeypatching*. Monkeypatches replace part of an existing object to alter that part's behavior. I demonstrated this technique using PyMock's `override()` function extensively in the latter half of the chapter.

Using TDD leads to higher test coverage, but it's easy to backslide. Programmers tend to be optimists (if you weren't, then you'd go insane), so they tend to overestimate test coverage. The next chapter examines tools for measuring test coverage, and how to deploy them in your development environment.

Consistent style is also important if multiple people are working on a body of code. The next chapter also examines how to enforce these stylistic guidelines through Subversion pre-commit triggers. In general, developers want high test coverage and consistent style, but without feedback, errors creep into the work of even the most diligent. Automation can provide much of this feedback.

CHAPTER 8

Everybody Needs Feedback

Developers want to write good code. However, their code tends to be worse than they believe it is. They think their tests cover more cases than they really do, and they believe that more of the code is exercised than really is. They tend to believe that they understand the code better than they really do, and they believe they produce fewer bugs than they do.

This is because programmers are by and large healthy optimists. They have to be. Truly understanding all the details in even a simple program requires years of study and experience, so they gloss over most of the details.[1]

This isn't some kind of strange, aberrant behavior. Over and over again, psychological research has shown that normal, happy people believe that they have more control than they really do. Depressed people seem to have an absolutely accurate view of the control they have over situations. Does that make them better programmers? Probably not. The depressed tend to be less creative, and they have a really hard time motivating themselves. On balance, it's better to be a healthy and functional human being, even if it leads to objectively unjustified optimism.

An experienced programmer does, however, tend toward cynicism. Experience hopefully brings an understanding of one's faults and shortcomings. It's not necessary to conquer your faults, but it is necessary to see them and work with them. The first step is getting the feedback to understand what those flaws are. A story illustrates this.

Years ago I had a dear girlfriend. She was one of the most brilliant people I've ever met. She was nearing the end of her doctorate in computational molecular virology. (After that, she headed off to veterinary school.)

There was a problem in her lab. Biochemists label things left and right—centrifuge tubes, test tubes, beakers, Eppendorf tubes, and so on. Getting anything done in a lab requires a Sharpie—a kind of indelible magic marker. And someone in her lab was stealing all the Sharpies. Whenever she needed one, she'd have to go questing for markers, stalling her lab work, and derailing her train of thought.

She railed endlessly to anyone who would listen about the inconsiderate thief who was stealing all the lab's Sharpies. She loved venting about it. She couldn't figure out who it was either, which made it all the more mysterious.

---

1. One of my favorite interview questions for system administrators is "Describe what happens when you type `telnet www.google.com`." No matter how deep someone goes, you can always ask more detailed questions.

One evening she opened up a desk drawer in her bedroom, and she tossed a pen in. Unlike most days, she looked down at the drawer. She broke down laughing, and she brought me over to look at it. It was full of Sharpies. Tens upon tens, perhaps hundreds of them.

She was the Sharpie thief. Every day she left the lab with two or three Sharpies in her pants. Every day she returned home, mindlessly opened the drawer, tossed the Sharpies in, and closed it. (Her organizational instincts were incredible.) She never noticed doing it. With this discovery, the Sharpies stopped vanishing from the lab.

The moral of this story is that feedback is incredibly important. Without the appropriate feedback, she might never have realized that she was the source of the problem in the lab. Without feedback, you often can't see your own faults.

What do developers need feedback for? Well, developers have their own drawers of Sharpies. Each person has errors they tend to commit. I double space after periods, and I have to go back through my documents pulling them out. My editor appreciates that. I also have a tendency to write overly complicated and general code. I have to strive for simplicity. I have trouble choosing appropriate names, and my comments often lack enough depth. I tend to be either too pedantic or not pedantic enough. Sometimes I use tabs by reflex, and my lines tend to be way over 80 characters long. I tend to miss simple error checks, and I like mock objects too much. I have to keep an eye out for these things. It's good to have tools, procedures, and an environment that help to prevent these from happening.

This chapter looks at several measures of quality. Some are quantitative and some are qualitative. Among the qualitative measures are coding standards.

Fundamentally, there are two kinds of feedback for development. They are social and environmental. *Social feedback* includes structured criticism through procedures such as code reviews, and it includes cultural norms such as interpersonal communication patterns and documentation habits. Rewards are also a kind of explicit social feedback, and I'll talk a little bit about them.

Environmental feedback encompasses technological gadgetry. Your project's tooling should give you feedback where social feedback fails. It can produce precise, focused, and immediate feedback on small things:

- IDEs and compilers let you know when code is syntactically broken.
- The source code repository can check for malformed code and refuse to accept submissions.
- The build system can fail the build when conditions aren't met, which I've already demonstrated in connection with unit tests in previous chapters.

This is all very important because it affects software quality. Software quality is about keeping errors down while making the remaining errors easy to find. Put another way, it is about making software that is easy to maintain without introducing new errors.

There has been a great deal of research into the kinds of errors that developers make. Different studies report different results, and it's hard to come to a firm consensus. Some consider the hard numbers produced in this area to be highly suspect. Much effort has been focused on classifying bugs and their relative frequencies, and some general themes have been revealed.

The scope of most errors appears to be limited. Many are outside the domain of construction. Most are the programmer's fault, and a lot of those are typos and misspellings.

A recurrent theme is failure to understand the problem domain and the design of the software itself. Happily, most errors seem to be easy to fix.

One plausible reason for the difference in quantifiable results between studies is that different environments, both social and technological, lead to different errors. The individuals in the mix probably contribute, too, so it is important to build on your organization's experience. I suspect that collecting per-user and per-group information to build targeted defect profiles is an area that is ripe for research and/or commercialization.

There are some practices that make errors easy to find. The first of these is an extensive suite of tests, which I've already discussed in previous chapters. Tests provide feedback, but there is further feedback about the quality of those tests, which is explored here.

Simple design, a core agile practice, focuses on building only the minimal functionality that allows the program to meet the user's needs. There are measures that successfully capture and quantify various aspects of a program's complexity.

Writing clear code helps to pinpoint errors. Clear code is written with the intention that it will be read.[2] It focuses on communicating intent to the user, with the computer as a secondary concern. Various tools assist in writing clear code. They check conformance with coding standards and consistency of style.

Stylistic consistency is one of the hallmarks of easily read code. In such code, names are chosen well, and they are chosen in a way that reflects the underlying system metaphor. Those names and the choices they embody are propagated throughout the code base. Typographical conventions are the same throughout, blocks are indented the same way in the same situations, spaces are added or omitted in the same manner, and so on. These choices are made in a way that is both simple and self-consistent.

While tools can help with some aspects of these practices, human eyeballs and procedural or cultural practices are often the best ways of helping to achieve these goals. The problem with tools is determining which aspects of these practices can be measured.

Measuring Software Quality

Measurements give you feedback. Quantitative measures give you precise numbers characterizing an attribute, while qualitative measures describe the general properties of the subject you're studying. They tell you what you have, but not how much of it.

Quantitative measures are appealing in that they can often be automated. They tell you a precise value of a specific attribute, but their specificity limits their utility. The results can be rendered graphically, making them favorites for management. (There are some people who fall in love with anything that you put in a spreadsheet.) They invite abuse at times, and in the wrong hands they render discussion moot, even if there is a point to be discussed.

Qualitative measures are much fuzzier, but they can often lead to greater insight. They are judgments such as "the code stinks," or "the style is awful." They constrain the mind less, and their contemplation often leads to ideas for quantitative measures. Qualitative measures don't lend themselves to automation, so those with a penchant for automation often give them short shrift.

---

2. As Tom Welsh (my editor) said, "Indeed, the more successful your code, the more times it will be read—and by more people."

Measurements

With any measure, the first question is "What are we trying to measure?" There are several factors characterizing the measure:

There are *attributes* and *instruments*. The underlying phenomena may be characterized by attributes that can be measured. Those attributes are determined, and the instruments of measurement are decided upon.

The instruments' results must be *reported*. A means of storage and presentation must be decided upon. They may be dumped into a database and analyzed, or they may be spit to `sys.stderr`. The means of presentation doesn't have to be fancy, it just has to be effective.

Often you measure to *effect change*. Do the chosen measurements provide effective feedback? Do the measurements of code complexity result in less complex code? Does a measure of test coverage result in better test coverage? Does it tell you where the poor-quality code resides?

Measurements often have *side effects*. Are your programmers now competing to see who can get the highest cyclomatic complexity number? Are programmers just adding tests to increase coverage instead of really testing the code? Will this cause the measure to lose its effectiveness at identifying poor-quality code?

Before you begin measuring, there are some fundamental questions for which the answers need to be understood:

What is the purpose of the measurement? What are you trying to accomplish? Is this for your own use, or is it intended to change the way everyone codes? If the measurement is for your own use, then the variance may be high, and the technique doesn't need much justification; you can be sloppy. If the measurement is intended to change the way everyone codes, then you need to choose a well-understood measure, and you need to do it in a consistent manner, as you'll need to justify your choices.

What is the scope? How widely will this measurement be used? The wider it is applied, the more impact it may have, both through positive control effects and unintended side effects.

What attributes are being measured? Imprecise ideas about what is being measured are likely to yield imprecise results.

What are the units? Unless you understand the units, you can't determine how it relates to other quantifiable values. Measuring an amoeba in feet is nearly useless. Measuring an elephant in angstroms is meaningless (although it does bring up the interesting question of where the elephant begins and ends).

What is the variability of the attribute? Unless you understand the variability of a measure, you can't determine how accurate your measurement is.

What is the measuring instrument? Don't use a micrometer to measure an elephant. Don't use a yardstick to measure an amoeba. Don't use line counts to measure program complexity. (Don't use a bathroom scale with an elephant either. It breaks.)

What are the units of the instrument? This ties in with the previous question. The units of the instrument must be compatible with the units of the attribute.

What is the reading's variability? Most instruments are imperfect. They have errors. Network problems cause sampling problems with remote probes. Statistical profilers can only give approximate usage reports.

How do the measurements and attributes interact? Retrieving page counts from a web server by making an HTTP connection increases the number of hits. For small, low-traffic web sites, this could be a problem. Measuring code coverage through execution affects how quickly tests run. Timing tests might exhibit failures while coverage is being examined.

What are the foreseeable side effects? Are the page hits artificially inflated? Are the timing tests dying mysteriously? Will reporting cyclomatic complexity result in an obfuscated code competition? Will reporting code coverage cause code coverage to improve? Will you be fired for stepping on your boss's turf?

Quantitative Measurements: How Much Is That Doggie in the Window?

There are common quantitative measures to which you've probably been exposed. These include the following:

- Test coverage
- Source lines of code (SLOC)
- Cyclomatic complexity
- Churn
- Recorded defect rates
- Development velocity

We'll be looking at three of these in detail: coverage, cyclomatic complexity, and development velocity.

Code Coverage

Code coverage is a family of measurements. There are many different kinds of code coverage. Cem Kaner covers 101 of them in his paper "Software Negligence and Testing Coverage" (`www.kaner.com/coverage.htm`). With 100 percent statement coverage, all statements in a program have been executed. This is not to say that all expressions have been executed. It is also not the same as saying that all branches have been executed. An if-then-else statement has been executed even if only the else block has been traversed.

With 100 percent branch coverage, every branch of every statement has been executed. In an if-then-else statement, both the then block and the else block have been traversed. Branch coverage is a much stronger metric, but it still doesn't guarantee that all expressions have been evaluated. In some definitions, a short-circuit logical operator is considered to have been executed even if the second operand has never been evaluated.

Branch coverage is appealing in several ways. It is easy to count. In many languages, the tracing mechanism can be used to obtain this number. It is unambiguous. When you say that 70 percent of statements have been executed, little further explanation is needed. The ease with which it is explained is part of its appeal. Anyone can grasp it in a moment.

Those factors make branch coverage seductive. There is a temptation to see it as a goal, but it is not. It is simply a tool, and like any other tool it has limitations.

Branch coverage tells you nothing about data flow. It doesn't tell you that a variable has never been initialized, that a constant is returned instead of the value your code spent hours calculating, or that an invariant value is being rewritten every time a loop is entered.

One hundred percent branch coverage only covers those branches that have been written, so it doesn't cover sins of omission. Necessary, but unwritten, code is invisible to this metric. According to one survey study, these kinds of errors account for between 22 and 54 percent of all bugs.[3]

Weak tests may hit all statements, but they don't hit them very hard. The tests don't exercise every predicate in the conditional clauses. Loops are only executed once, and many bugs don't occur until they're executed several times. Default values are modified, and the new values leak into subsequent calls, but the test framework clears them every time.

Mock objects short-circuit interactions between methods. In Python, they allow complete isolation, so it's possible that the real function is never called. Although the statement has been executed, it hasn't been executed with real data.

Branch coverage doesn't report errors that take a while to manifest. It doesn't catch environmental interactions. Table-driven code is inscrutable to branch coverage tools, which miss all of the embedded logic. They miss any place where work is done in data instead of code, and branch coverage completely misses the interactions between interrupts and signal handlers.

With all these problems, why use branch coverage? Because it yields useful information; but you have to be aware of that information's limitations. If you have low test coverage, then you probably have a problem. You should look at where the test coverage is missing, and then decide if it should be addressed. If it's old code that's well debugged and rarely changes, then it's probably not worth focusing efforts there. If it's in highly defective code, or code with a high churn, then it might be worth focusing testing efforts there.

What constitutes low test coverage? Below 85 percent is a number that's bandied about, but there seems to be little academic basis for it. It may be a number that someone picked out of a hat at some point and has been referenced ever since, like an urban legend.

What is the branch statement coverage of your code? Unless you're measuring it, you're probably overestimating it. Typically, unit tests only cover 50 to 60 percent of the code in a system. You can probably look at my code and get a feel for the coverage, but when you look at

---

3. Brian Marick, "Faults of Omission," *Software Testing and Quality Engineering Magazine* (January, 2000), `www.testing.com/writings/omissions.pdf`.

your own code your estimate will be too high. People tend to have a blind spot when it comes to their own weaknesses. There is likely to be a moment of shock the first time you wire up one of the tools described later in this chapter.

Some have an aversion to measurement. There's not really an excuse for this. Refusing to measure when you have the tools available means that you are willfully ignorant, but there are reasons to tread carefully. Measurement has motivational effects, and these effects can be good or bad. People tend to optimize for anything that they are being judged by. People like to look busy and productive, so measure and report carefully.

Complexity Measurements

As with code coverage measurements, there are many different kinds of complexity measurements. The example we'll be wiring up later is called cyclomatic complexity, or McCabe complexity, developed by Thomas McCabe in 1976. The measure was almost a side effect of the paper's larger achievement of defining what an "unstructured program" is. It determines complexity on a per-function or per-member basis by examining a program's control flow graph.

A control flow graph pictorially represents how execution passes through a program. In these diagrams, statements that don't affect execution paths are ignored. Only those statements that entail decision points are included. The flow graph for the following program is shown in Figure 8-1.

```
def foo(x):
    while x > 5:
        if x < 2:
           print "a"
        else:
           print "b"
```

The cyclomatic complexity algorithm adds a link from the end of the program to the beginning. This is shown as a dotted line in Figure 8-1.

In hard mathematical terms, cyclomatic complexity is the smallest number of linearly independent paths that it takes to span the flow graph. A set of partial paths span a graph when every possible path through the graph can be described using a combination of these partial paths. A set of paths that span this graph is shown in Figure 8-2. Every path through this graph can be described by combining these five graphs.

The next part of the definition is *linear independence*. If you've had a linear algebra class, this should be familiar. The paths through the graph are linearly independent if there is no way to combine all of them at the same time in such a way that they cancel each other out.

Imagine that you're dropping breadcrumbs as you walk one of the partial graphs. Then you try walking one of the other paths that connect to this one, and you continue doing this until you've walked through all of the partial paths. If you can pick up all the breadcrumbs you dropped, then your paths were not linearly independent.

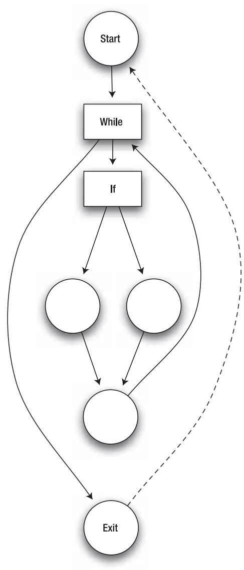


Figure 8-1. *The control flow for an if-then-else statement inside a while loop*

In Figure 8-2, the three paths on the right (3, 4, and 5) are not linearly independent. You can walk path 3 and then path 4 dropping breadcrumbs along the way, and you'll end up at the beginning. Then you can follow path 5 backward, picking up the breadcrumbs as you go, and you'll end up at the beginning having collected all the crumbs.

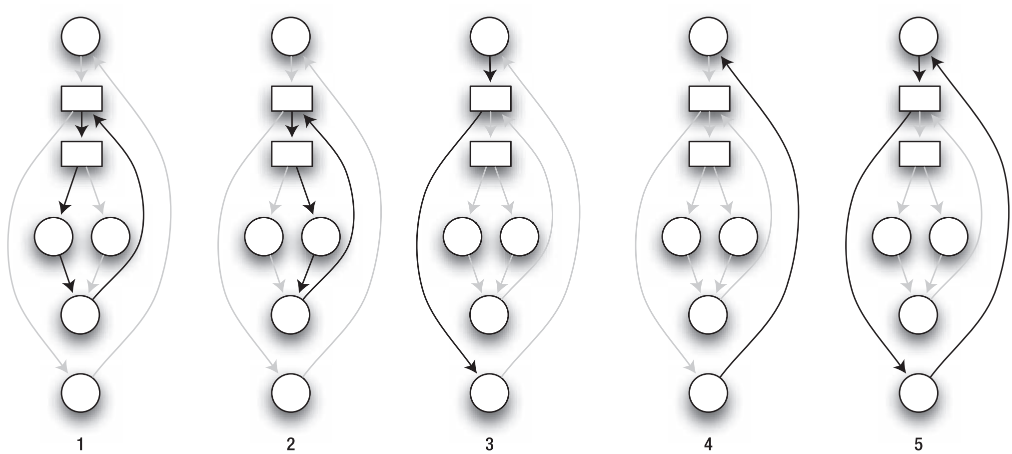


Figure 8-2. *A spanning set that is not linearly independent*

Figure 8-3 shows a linearly independent set of partial paths that span the graph. This is not the only one possible, but three is the smallest possible number for this graph. No set of two partial paths can be combined to describe all possible paths, and no complete set of four

or more partial paths is linearly independent if it spans the graph. Every closed directed graph can be characterized like this.

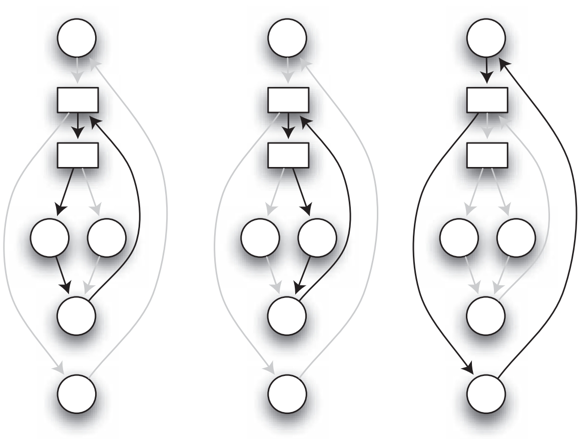


Figure 8-3. *A linearly independent minimal spanning set*

So that's a description of what cyclomatic complexity really means. It's how many different control flows are in a section of code. You could calculate it by drawing graphs and figuring out the spanning sets, but there's a much easier shorthand:

```
C = 1 + number of decisions
```

There is always at least one straight line through any graph, so cyclomatic code complexity starts with one. Each decision point adds another possible path. Some constructs add more than one. Every time a construct is encountered, the complexity index is increased by a specified amount. A simple calculation might use Table 8-1.

Table 8-1. *Sample Scores for Use in Calculating Cylcomatic Code Complexity*

Construct	Effect	Reason
If	+1	A decision is made.
Each elif	+1	A decision is made.
If-else	+1	A decision is made. (A plain If statement has an implicit, but empty else clause.)
While	+1	A decision is made.
For	+1	A decision is made.
Try	+1	Exceptions generate a new flow of control.
First except	0	The first is already accounted for by the try block.
Subsequence excepts	+1	A new choice is added.
Finally	0	All paths just rejoin here.
With	0	No control flow is visible to the routine.
Decorators	0	There is no alteration of the program flow.

There is more than one way to derive the cyclomatic complexity for a routine. The differences are based primarily on the control flow graph that is generated. Each logical comparison in an if-then or while statement can be viewed as generating an alternate condition. This can potentially result in much higher cyclomatic complexity numbers.

Generally, the lower the cyclomatic complexity, the better. Values in the range of 1 to 5 are considered to be trivial. Values from 6 to 10 are considered to be low risk. Numbers between 11 and 20 signify moderate risk. Numbers between 21 and 50 are considered high risk. At 50, you should consider submitting your routine to an obfuscated code contest rather than your source repository. The word *untestable* is often used in this context. These cutoffs are somewhat arbitrary, and as far as I know, there is no basis for their use other than experience and informed opinion.

Velocity: When Are We Done?

Velocity is a quantitative metric describing how much work a group can accomplish in a given time. Velocity is the primary measure for capacity estimation in most agile methodologies. It is most frequently used in development environments with well-defined iterations.

At the beginning of each iteration, the tasks available are placed on a board. Together, all the developers assign an effort estimate to each task. The estimates come from a small set of possible choices that correspond to point values. At the end of the iteration, the team sees how many points of work they've completed. The team's velocity is the number of points completed divided by the number of days worked.[4]

The first time through, the team is flying blind. They can make the estimates, but they can't convert those work estimates to time estimates. Velocity provides this conversion.

In successive iterations, the previous velocity measurements are combined to produce an average velocity, and this value should become more accurate over time. As the accuracy improves, the team can use this number to reschedule development or drop features as appropriate.

There are different methods for assigning estimates. Some use raw points, and some map between a natural language scale and points (e.g., small, medium, large, and that's-too-big-to-estimate). No matter what the details, they all use a small set of values, often no more than four or five.

Next, I'll describe two scales I've had direct experience with. One scale uses a raw point range of 0 to 3. A 0-point job is trivial. A 3-point job should probably be broken into smaller pieces. The numbers in this scale are not linear—a 2-point job takes much more than twice as much effort as a 1-point job, and 1-point job takes much more effort than a 0-point job.

Another scale uses the sizes extra small, small, medium, large, and epic. At one end of the spectrum are extra-small tasks, which are trivial, and at the other end are epic tasks, which are inestimable and need to be broken down into more manageable chunks. The rationale for using sizes rather than point values is that sizes can be mapped to a nonlinear scale, so that small might be 1 point and large might be 8.

---

4. Here the units are points per day. Always know your units.

The scrum methodology uses direct time estimates in hours, in which no task takes more than a day. All of these can be interchanged to some degree.

These values are purposefully fuzzy. Each group's definition will be a little different. What matters is that the team is consistent. Over time, the velocity calculations—whether in points, effort, or hours—become more accurate. The team works in small enough increments that daily stand-up meetings and periodic sprint retrospectives give them timely feedback, and this allows for improvements in estimation.

Qualitative Measurements: It's a Shih Tzu!

We are capricious beasts, and we are rarely as rational as we'd like to believe. Often, qualitative measures are the things that truly matter to us. I can have the best job in the world, but something goes wrong. My manager changes, and nothing else really changes about my job. The new guy is personable, in fact downright likable. By any measure of wage or work hours, or a listing of responsibilities, my job has remained the same, but suddenly I hate it. Getting to work is a chore. I'm constantly stressed. My ability to complete work declines.

Something has changed, but I can't say what. I can't measure the cause, but it's real and it matters. There has been a qualitative change, and its ruining my job.

IS THAT REALLY A MEASUREMENT?

Somewhere along the line, I was asked if qualitative judgment could really be called a measurement. I have a background in biochemistry, and I spent a small chunk of my life in a lab. Lab notebooks were full of qualitative measurements like this:

Tube A: Clear
Tube B: Cloudy
Tube C: Kind of murky
Tube D: Completely opaque

Each one of those is a measurement. You're determining some kind of data and recording it. You can describe code similarly:

Function A: Terrible code
Function B: Not too bad
Function C: Pretty good
Function D: Obviously Noah Friedman [5]

So I think it's fair to say that readability, continuity, and elegance are measurements, even if they don't have an obvious numerical representation. It's the systematic recording that makes something a measurement.

5. Noah Friedman wrote large chunks of Emacs. People who read his code have been known to laugh out loud with pleasure. My programming skill is measured in millifriedmans.

In the same way, there are many qualitative measures related to software. These are the things that you feel with your gut. These are the elements of judgment. When you read a program listing and you smile at the cleverness and clarity of the bounds checking, that's a qualitative judgment. In the same way, the sudden feeling of revulsion when you look at Bugzilla's code is also a qualitative judgment. These judgments are the measures of appropriateness of naming, agreement among those names, and elegance of control flow.

Qualitative changes often have quantitative effects. Poor architecture leads to a lower velocity, as do many other flawed development aspects. One of these is readability. Code that is hard to read is hard to modify. Code that is inconsistent is harder to read. The need for consistency leads to coding conventions.

Coding Conventions

There are three primary aspects to coding conventions. The typographical standards dictate the code's appearance. How many spaces is each block indented? Do spaces bracket the equal sign in an assignment? How about in a keyword assignment?

Naming conventions determine how names for variables, classes, and methods are chosen. They provide a common grammar and map the system metaphor into the programming language.

Structural conventions determine how a project is laid out. They determine where data and documentation can be found, and where the code and tests live. They provide a structure so that both developers and tools can examine the code base.

Coding conventions supply your project with a common language. They allow you to take more for granted. You don't have to decide where a file will go. You only have to decide what kind of a file it is. The convention supplies the location.

Coding conventions help to transfer knowledge across projects. When multiple projects share the same conventions, the developers can use each other's knowledge. They know where the unit tests are. They know that `ViewInterface` names a class rather than a method, and that `view_interface` defines a method or variable.

This consistency allows developers to learn new code more quickly.

Naming standards reduce name proliferation. A `linear_transformation` could reasonably be called a `linear_matrix`, a `linear_transform`, a `transform`, a `scaling_matrix`, or a `rotation`. Choosing one leaves less to remember.

Naming standards can compensate for language weaknesses. Unlike Java, Python doesn't provide interfaces as a language feature, but they can be simulated using various techniques. The standard can declare that classes used as interfaces shall be given names like `FooInterface`. Any time a developer sees `BarInterface`, they will know how it is being used. This lets developers bring along a useful feature from one language to another without formal support in the new language.

Naming standards can also emphasize the relationships between items. Declaring that collections must have plural names, and that each element in a collection must be referred to by the singular of that collection's name, instantly gives the user reading a loop a clear idea of the variables' relationships.

Almost any convention is better than none. Convention can be understood as a gradient, and the more that is specified, the less has to be deduced. Conversely, the less that is specified, the more must be deduced, and each deduction takes time and mental energy. Any reduction in variety will help.

The principle behind all this is that *structure is good*. Some argue that choice is a good thing, and it is, but only up to a point. Beyond that point, people are overwhelmed by choice. Too many options at a store actually decrease the chances that a customer will buy something. Our minds can't cope with any more information. It just becomes mental noise. Coding conventions reduce choice through clear guidelines.

This book demonstrates the utility of conventions—typographical, naming, and structural. When you see the font and location, you instantly know that this is the main narrative. When you see a double-lined section, you know it's a warning. When you see a sidebar, you know that it might be interesting, but that it's not essential. Imagine what this book would be like without consistent conventions.

Consistency is a good thing, but some consistencies are more important than others. Consistency with the world in general is good, but consistency within a project is more important, and consistency within one package is more important still. In turn, consistency within a module is more important than within a package, consistency within a class is more important than within the module, and consistency within a single function or method is paramount.

In general, when it comes to coding standards, questions are important. If you have questions, ask your partner. If she's not available, ask your teammates. Open and instant lines of communication are valuable. Physical proximity is always best, but often not possible. IRC or other shared-channel chat protocols are a distant second. Person-to-person chat is less useful. Hiking across the office is bad, and the phone is even worse. Sometimes, though, it may be the only choice.

There are very practical benefits conferred by consistent code conventions. When applied, they reduce churn due to reformatting, and diffs between file revisions become smaller, saving effort when integrating.

There are a variety of options for specifying coding conventions. At one end are flexible guidelines that are suited for very small groups or very large groups with diverse personalities. These may be little more than a collection of suggestions. At the other end are extremely detailed documents describing every *t* and *i* to be crossed and dotted. In either case, these are often best implemented as (or in conjunction with) a set of examples demonstrating the desired standards.

Coding conventions are desirable when the code is maintained or extended. In practical terms, this implies that anything other than disposable code for your own use should adhere to a well-defined coding convention. Certainly all agile projects fall into this category. Pair programming automatically involves review, and a well-defined system metaphor lays the foundations of a naming convention that should be extended through the entire code base.

The definition of good code varies from person to person, but a group needs a shared definition. Where should this shared definition come from? Answering this question requires some consideration of how software projects work.

Software projects depend on two kinds of authority: organizational authority and technical authority. The two are independent. *Organizational authority* comes from the structure of the company. *Technical authority* comes from experience and intelligence. Technical decisions need to be made with respect to technical authority.

Programmers often tend to despise management. Management, in turn, often disregards the programmer's expertise. But both groups contribute to the larger goals of the company, and each needs the other's know-how. Only when they respect each other and listen to each other can a project reach its maximum potential.

In all cases, managers should be involved with development to clearly communicate the company's goals. The agile practice of identifying a well-defined customer addresses part of this. The customer is brought into the development cycle and works closely with the developers. Short iterations with a predictable workload provide management with the feedback they need to understand development's progress.

So where do the coding conventions come from? If a coding style declaration comes from nontechnical management on the basis of organizational authority, then it will be rejected, so it must come from a position of technical authority within the development organization.

Several things make a coding standard work. First and foremost, the coding standard must be disseminated and understood. Clear examples of the coding standard must be available.

The code must be considered to be a shared asset, and everyone should expect their code to be read. Moreover, this expectation should be true. Every line should be seen by another pair of eyeballs.

All code should be seen by multiple people. There are aspects of a coding standard that can be verified by machine, all relating to typographical structure. The true meat must be verified by people, and that means people who weren't the sole author. Such outsiders don't have the same blind spot for the author's foibles.

When the code is submitted, it should meet quality guidelines. Some argue that the code should be signed off by those with more technical authority. There is a range of views as to how this should be done. At one end are those that suspect the programmer's competence and argue that all submissions must be approved. At another point along this continuum are organizations in which only one sizable chunk of code must be submitted for approval. Once the programmer has passed the "this is how we program here" test, they are allowed to submit code on their own. At the far end of the scale are those who implicitly trust those whom they hire without verification, which is a situation I have personally seen lead to bad code.

It has been suggested that if a project has a technical manager, then this manager should read all code. If the manager can't read the code, then it needs to be rewritten. In such cases, it is a boon to the project if the manager isn't a hotshot programmer, and the authors have to write down to his level.

There are different degrees of formality for coding standards. The degree of formality is directly related to the number of people on a project. The standard should be as informal as possible. Established projects should use the guidelines that they already have. If they don't have any, then guidelines should be established. Those guidelines should match the code base's existing style whenever possible.

Occasionally you'll have to break style. This is done primarily when the rules make code less readable, even for someone who is knowledgeable about the rules. This should be a rare occasion. The other reason is to be consistent with extant code that already breaks the rules, although in these cases, fixing the offending code is often a better solution. The longer it incubates, the further the stylistic infection is likely to spread through your code base's healthy tissues.

Welcome Back to Python

Unlike many other languages, Python has a natural authority for coding standards. This is Guido van Rossum, Python's creator. He is sometimes referred to as the Benevolent Dictator for Life (BDFL).

His wisdom is encapsulated in several Python Enhancement Proposals (PEPs). These conventions were laid out early in Python's development, so they're reasonably well disseminated. There are five documents related to coding conventions:

- PEP 7: Style Guide for C Code
- PEP 8: Style Guide for Python Code
- PEP 20: The Zen of Python
- PEP 257: Docstring Conventions
- PEP 287: reStructuredText Docstring Format

The first (PEP 7) relates only to C code, so I'm ignoring it in these discussions. The last two (PEPs 257 and 287) relate to docstring formatting. reStructuredText (RST) is a simple markup language similar to that used by many wikis. It is used for complicated docstrings or when the code is intended to be self-documenting.

PEPs 8 and 20 are the real meat. PEP 20 explains the philosophy behind many of the decisions made in the evolution of the language, and it helps in understanding why things are the way they are. It's written by Tim Peters, but he's channeling the BDFL. It's the best starting place. In fact, PEP 20 is embedded within the Python interpreter:

```
>>> import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea—let's do more of those!
```

This contrasts strongly with other languages that offer many ways of performing the same task. Here there's only one way to do it.

PEP 8 is the global coding standard. All groups should attempt to adhere to it. Python is refreshing (some might say maddening) in its approach to indenting. In most languages, the placement of block delimiters results in a variety of gross layout decisions. Since Python uses leading whitespace alone, there are no block delimiters to inspire theological debate.

PEP 8's summation of block formatting boils down to this: each level is indented four spaces—don't use tabs. The document is well written, and it is available at `www.python.org/dev/peps/pep-0008/`.

Naming is the area where Python is least consistent. The styles advocated in PEP 8 are shown in Table 8-2. For the most part, new packages adhere to these. Unfortunately, the recommendations were taken up long after a variety of naming styles had infiltrated the core libraries. There are now aberrations like the string buffer I/O module `StringIO.StringIO`. Attribute names using camel case with an initial lower capital (à la Java) are often seen, but this style is advised against. Still, it's hard to escape this style because several notable packages, among them SQLObject, use it.

PEP 8 only addresses the most basic aspects of writing docstrings. PEP 257 fills in the details, and it's well worth the read. It explains conventions for formatting docstrings. PEP 258 goes even further. It defines a simple and attractive markup language for use in Python docstrings and documentation.

Table 8-2. *Python Typographical Naming Conventions*

Entity	Name Convention	Example
Package	Short lowercase underscore separated	`my_package`
Class	Camel case	`MyClass`
Attribute	Lowercase underscore separated (LCUS)	`my_var`
Function	LCUS	`my_func`
Private class	Camel case with leading underscore	`_MyPrivateClass`
Private attribute	LCUS with leading underscore	`_my_private_var`
Private function	LCUS with leading underscore	`_my_private_func`
Really private attribute	LCUS with two leading underscores	`__my_var`

All of these standards address the typographical aspects of coding conventions, but they don't deal with the semantic aspects of naming. To some extent, I've tried to provide a basis for structural decisions in earlier chapters, but when it comes to the semantic aspects, you'll have to develop your own. The particulars of your naming scheme will depend heavily upon the system metaphor you develop in your project.

Never Try to Fix a Social Problem with a Technical Solution

Technical fixes to social issues are usually failures. They frustrate people. The people imposing the technical limitations pay for the failure, and they usually pay in the currency of respect.

Before people can comply with a standard, it must be available. A secret standard is useless, so it must be published, and it should be published in a well-known location. Often a

code catalog containing snippets is better than an English document. A catalog consisting of real code from the project is even better. New samples can be taken from the code base as part of a motivational system.

Rewarding good code is a mixed idea. It opens up the entire topic of rewards and penalties. Rewards are a form of measure, and like all measures they must be used carefully, for they are particularly prone to unintended side effects.

The goal of any reward should be increasing the productivity of the entire team. A reward that emphasizes individual achievement but undermines the team is worse than no reward at all. If individuals are rewarded, they should be rewarded for contributing to the achievement of group goals, and in such a way that anyone can achieve the reward by doing their job well. Choose goals that you want to amplify and that won't stomp on other goals.

If you are a manager and you don't understand what good code is, then don't make that judgment, for rewarded code should be exceptionally good. If it is not, then you'll look like an idiot. No matter what, the reward should not reflect any other attributes of the programmer receiving the award. If the programmer is an arrogant, foul-smelling antisemite working for an Israeli security firm, but he writes a chunk of code that makes people gasp at its beauty, then he gets the reward, no matter what the political consequences.

The rewards should be something the programmer wants. "Attaboy" rewards are distasteful. Rewards for the entire team for individual contributions are an interesting thought, but I have little experience with them. If you do, then I'd love to hear from you about them. The problem with individual rewards is that they are at some level in conflict with the principle of collective ownership.

Code Reviews

Code reviews are an extremely valuable tool for achieving code quality. They are among the most important forms of qualitative measurement, taking in all aspects of the code. Moreover, code reviews tie in very closely with agile principles. They fall into two broad categories: formal and informal reviews.

There is nothing magical about pairing. It involves a constant code review process. A second developer gets to comment on every line of code as it is written. Pairing provides a second set of eyeballs to help enforce coding standards. It's like taking a formal review and rolling it into the daily development process. Pairing does convey other social benefits, but they're not responsible for the primary quality effects.

There are some important things to remember about pairing. It shouldn't be done 100 percent of the time. There are some things best done solo. Researching new technologies, writing summaries, and just reviewing the code are often done best on an individual level.

Human interactions are a delicate thing. Most people in an office will be able to pair with most others, but there will almost certainly be some combinations that don't work well together. Some people may get on each other's nerves, be unable to communicate with each other, or just dislike each other in a deep, visceral way. These people shouldn't be forced to pair with each other. That is simply cruel and probably unproductive.

This is not to say that an individual who refuses to pair with anyone or offends everyone should be tolerated—but I am speaking specifically about particular pairwise interactions.

If a project doesn't use pair programming, then official code reviews should be instituted.

Renaming

Naming conventions help, but you (or others) will sometimes choose inappropriate names. Rename as soon as you realize that there is a better name. The longer a name remains in the code base, the more calcified it becomes. Other names are chosen to relate meaningfully and consistently with the extant names, and these names must also be changed when renaming.

When changes are limited in scope, refactoring tools are immensely useful. However, they can't find related names (yet), so they can't help locate all the interdependent names in a well-established project. However, they do make changing each name trivial, and making the early change becomes far less onerous. Get familiar with your IDE's renaming tools. Rename early, and rename far less often in aggregate.

Communication

Documentation should be available for the project. It doesn't have to be comprehensive, but it should guide developers through the overall system. Many errors are due to developers not understanding the problem domain, or the project's architecture and implementation. Reference materials for these should be available.

Written documentation is not sufficient for expertise in any field. A recipe prepared by Thomas Keller of French Laundry fame will taste very different from the same recipe prepared at home.[6] A great chef depends upon skills and judgments that have taken years to acquire. Doctors and veterinarians go through lengthy internships, and musicians spend endless hours practicing with each other. Programmers must communicate, too.

Curiosity is good. Questions should be encouraged, and opportunities for information exchange should be encouraged. Mentors help when a person first enters a company. Brown bag lunches on major subsystems are a good idea, too, but if I had my druthers the company would always supply the brown bags.[7] It's really a minimal expense compared to the salary of a skilled developer.

---

6. Thomas Keller is considered one of the best chefs, and his restaurant French Laundry is consistently listed among the top four in the world. See `http://en.wikipedia.org/wiki/Thomas_Keller`.

7. Brown bag lunches are (typically informal) talks given on the company premises. Often these happen at lunch time, and *brown bag* refers the fact that everyone brings their own lunch. Such arrangements are common at companies in the San Francisco Bay Area.

Technological Feedback: Bad Programmer, No Cookie

There are three places in the development cycle where tools can be used to provide feedback. At the keyboard, IDE or command-line tools can suggest changes according to coding conventions. When the defective code is committed, revision control hooks (a.k.a. triggers) can prevent it from reaching the code base or report questionable practices. When the code is built, analysis requiring the entire code base is performed. Running unit tests is one such example.

There are things that must never be allowed, and there are things that should not be done. The things you must never do can be prevented, and the things you should not do should raise warnings. Treating a “should not” as a “must never” is a crime. It maddens and frustrates people, and it makes your project unpleasant to work on.

There are some things that should never be allowed on a Python project. Some are typographical conventions. Unparsable code, leading tabs, inconsistent indenting, and questionable trailing whitespace should never be allowed into the code base. A build must never succeed if the unit tests fail, and significantly decreasing test coverage should never be allowed.

There are things that are looked down upon or that are indicative of other problems. Overly complex methods probably need to be simplified. Low code coverage suggests inadequate testing. Bad style probably indicates poorly reviewed or insufficiently thought-out code. These things should be checked and advised, but they should not trigger commit or build failures.

Coercion at the Keyboard

Pydev offers a variety of features to assist with writing and analyzing code. These features include the following:

- Autoformatting
- Code analysis (with Pydev)
- Templating
- Unit test execution

All of these features are configured from the Eclipse Preferences window. On the Mac, you open it by selecting Eclipse ➤ Preferences. On Windows, it is under Window ➤ Preferences.

Autoformatting is one of the most useful but least contemplated features. Pydev offers several settings, all of which have defaults conforming with PEP 8. The most important of these are the indentation settings, which are specified under General ➤ Editors ➤ Text Editors, as shown in Figure 8-4.

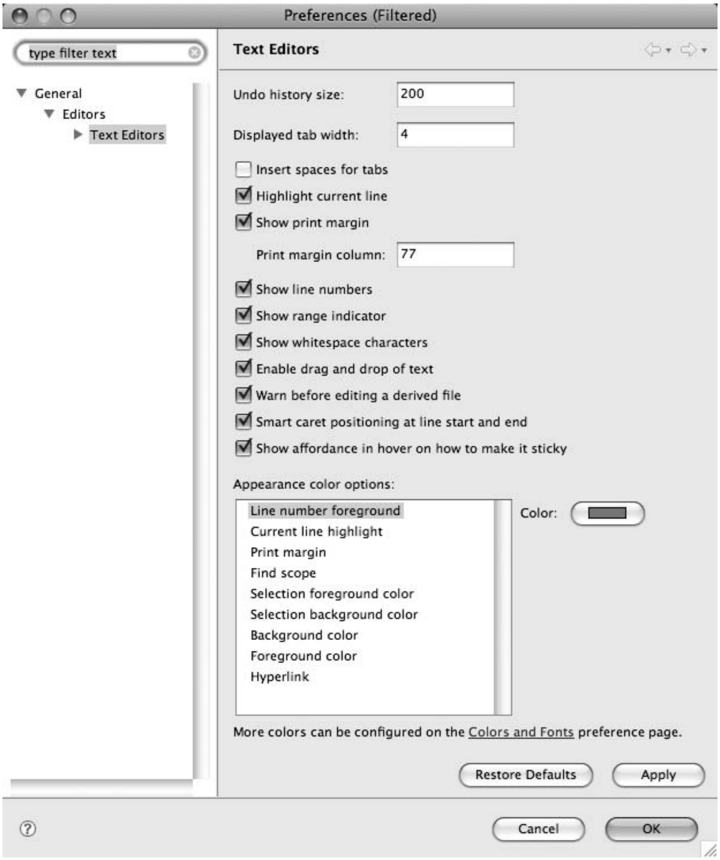


Figure 8-4. *Changing indentation settings in the editor properties*

To match PEP 8, "Displayed tab width" should be set to 4, and "Insert spaces for tabs" should be checked.

Under Pydev ➤ Code Style, there are four panels: Block Comments, Code Formatter, Docstrings, and File Types. These affect the way Pydev assists with code generation. It is clear how both Block Comments and Docstrings modify the formatting. Code Formatter only affects two changes: it determines if spaces are used after commas and if spaces are used both before and after parentheses. File Types determines which file name extensions the Pydev formatting tools operate, but these do not affect the association between the editor and Python files.

Under Pydev ➤ Typing, you will find more settings affecting the completions and formatting actions that Pydev performs. This panel is shown in Figure 8-5.

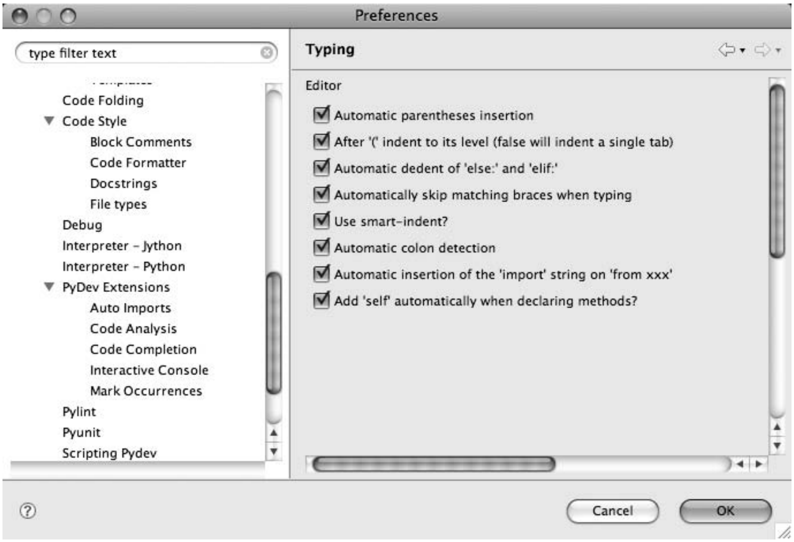


Figure 8-5. *The Pydev typing settings*

The commercial Pydev extensions add a significant number of code analysis features. These are accessed from Pydev ➤ PyDev Extensions ➤ Code Analysis. The tabs available are Options, Unused, Undefined, Imports, and Others. The odd man out is Options, which specifies when the analyzer runs. The rest of the tabs allow control of specific aspects of the analyzer itself. Each feature listed can signal an error or a warning, or be ignored. The Unused tab is shown in Figure 8-6.

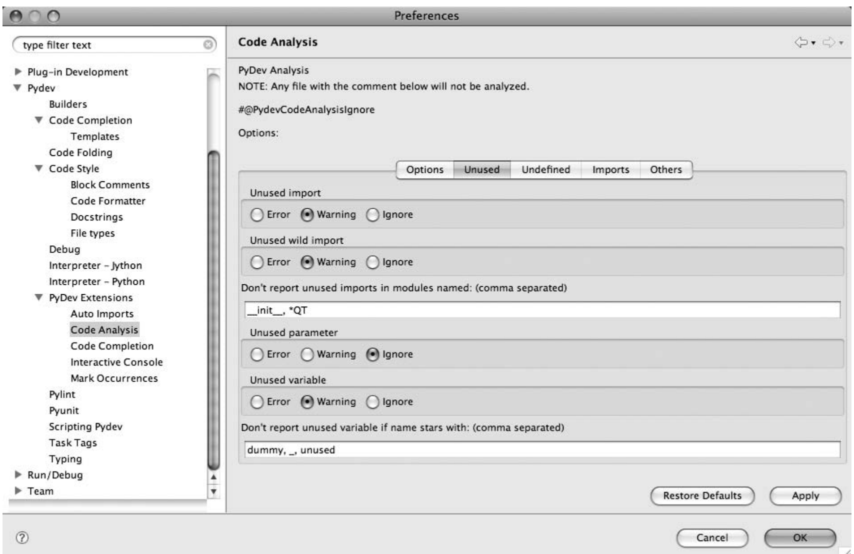


Figure 8-6. *The Unused tab on the Pydev Code Analysis panel*

The code analysis features provide the clearest examples of immediate feedback about errors in the code. These are only at the most rudimentary syntactic level, but they help identify issues that would otherwise be found when running the unit tests.

Templates are macro expansion mechanisms. When you press Ctrl+Enter, Pydev replaces the word you are currently typing with a template. Pydev defines macros for most Python control structures, but you can also define your own. The definitions can contain fields that must be supplied by the user.

In my environment, I have a template named `pym` that starts a PyMock Nose test. Templates are defined through the panel shown in Figure 8-7, who's path is Pydev ➤ Code Completion ➤ Templates.

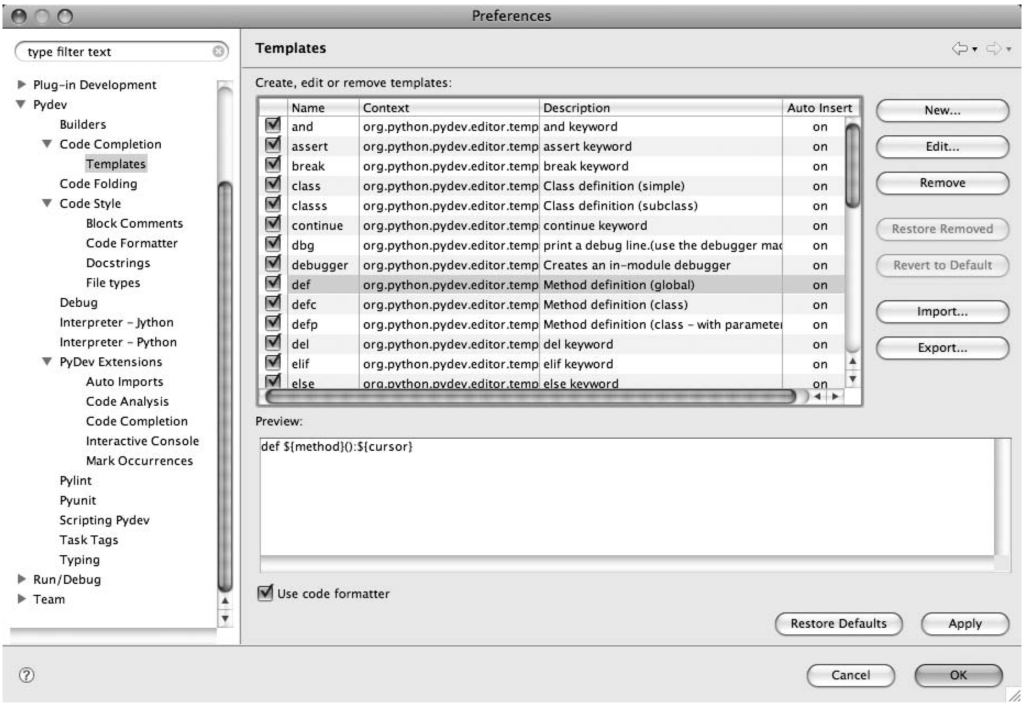


Figure 8-7. *The Pydev Templates panel showing the def macro*

The Edit button brings up the New Template window (shown in Figure 8-8). The Name field is the string used to choose the template when expansion happens. The Description field is just used in the Templates panel.

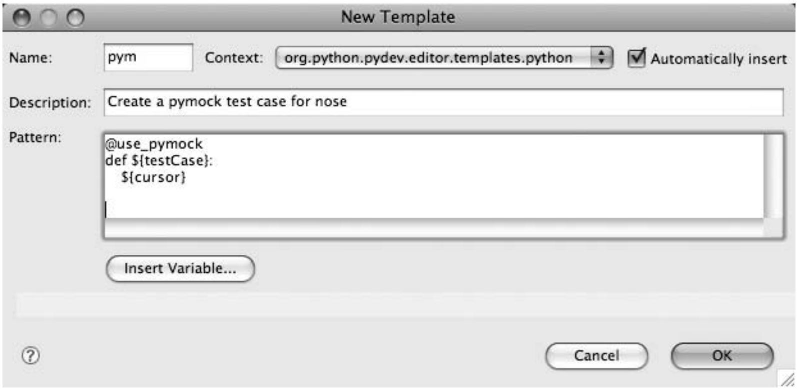


Figure 8-8. *Creating a new template*

The Pattern field defines the template body. Variables are denoted with the syntax `${foo}`. There are a handful of predefined variables that can be listed via the Insert Variable button. The most commonly used is `${cursor}`, which is the point the cursor will be left at when the macro expansion is complete. Unknown variables are filled in at expansion time, as shown in Figure 8-9.

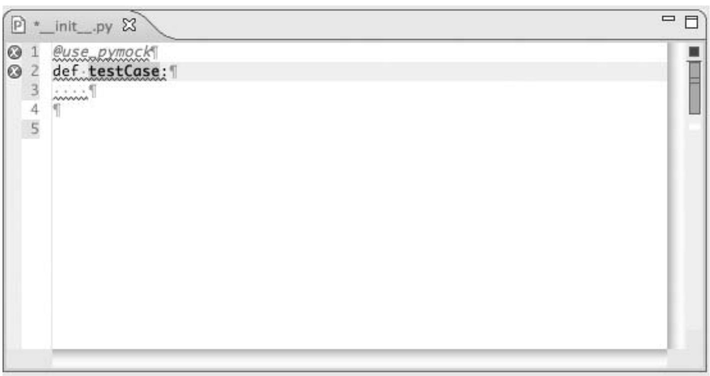


Figure 8-9. *The pym template being expanded*

The developer replaces `testCase` with the new method's name. When Enter is pressed, the focus passes to the next undefined variable. At the end, the cursor is placed where the cursor variable indicates. The templates can be exported and imported. A master templates file for the project should be exported to the root and checked in with the source, and it should be imported when it changes.

When Code Is Submitted

Subversion supplies hooks to validate committed code. These hooks happen after a transaction has been committed and the files have been copied to the server, but before the change has been committed. If the hook fails, the commit fails, and `stderr` is reported to the submitter.

This is the mechanism used to prevent atrociously malformed code from reaching the repository, and for supplying reports of offensive code.

To set this up, a `pre-commit` file is created in the Subversion `hooks` directory. On my development system, this is `/usr/local/svn/respos/hooks`. The file should be executable. If it is not, then it will fail. If the commit succeeds, this script returns a zero exit code. If it fails, the script returns a nonzero exit code and `stderr` is reported to the submitter. `stdout` is ignored. The precommit script acts as a dispatcher for other scripts that perform the real work.

Writing precommit hooks is a formulaic process, much the same in any language. The script must finish quickly to prevent the repository from blocking, so running a full compile to verify integrity is out of the question.

Precommit hooks are always passed the same two arguments: the repository path and the transaction number. These are used in conjunction with `svnlook` to get information about the transaction. The list of files is retrieved with `svnlook changes`. These files are iterated through, and the contents of each file are printed using `svnlook cat`. The log message can be accessed with `svnlook log`.

While the hooks should examine the code written in the project, there are some Python files that should be ignored. These are third-party files checked in to facilitate builds, so the precommit scripts must ignore these subtrees.

Amazingly, this hasn't been packaged up until now. I've done it as part of this chapter, however. The package is named svnhooks, and it can be installed with `easy_install`. It provides a simple framework for producing your own hooks. It can terminate the build or send notifications, and it comes with hooks to check for the following:

- Leading tabs
- Mismatched leading whitespace
- Windows line endings
- Trailing whitespace after \ at the end of the line
- Syntactically correct Python
- Suspiciously complex code via PyMetrics
- Questionable semantics via PyChecker or PyLint

PyChecker and PyLint are semantic verifiers. They check for constructions that are legal but questionable. Such things include redefined methods or locals overriding Python built-ins.

■ **Caution** Svnhooks supports PyLint, but the package is in questionable condition for use under Python 2.5. As of version 0.14.0, it won't correctly install unless the source is altered by hand.

My project's precommit script fails if indenting, syntax, or line lengths are invalid. It sends warnings to dev@sample.org if the code is too complex or if it fails the lint check. The file follows:

```
$ cat  /usr/local/svn/repos/hooks/pre-commit
```

```
#!/bin/sh

# PRE-COMMIT HOOK
#
# The pre-commit hook is invoked before a Subversion txn is
# committed. Subversion runs this hook by invoking a program
# (script, executable, binary, etc.) named 'pre-commit' (for which
# this file is a template), with the following ordered arguments:
#
#   [1] REPOS-PATH   (the path to this repository)
#   [2] TXN-NAME     (the name of the txn about to be committed)
#
# The default working directory for the invocation is undefined, so
# the program should set one explicitly if it cares.
#
# If the hook program exits with success, the txn is committed; but
# if it exits with failure (non-zero), the txn is aborted, no commit
# takes place, and STDERR is returned to the client. The hook
# program can use the 'svnlook' utility to help it examine the txn.
#

IGNORE='[^\/]+/(?!thirdparty)/.+'
ADDR='dev@sample.org'

/Users/svn/bin/whitespace_check "$REPOS" "$TXN" "$IGNORE" || exit 1
/Users/svn/bin/syntax_check "$REPOS" "$TXN" "$IGNORE" || exit 1
/Users/svn/bin/length_check "$REPOS" "$TXN" "$IGNORE" || exit 1
/Users/svn/bin/complexity_check "$REPOS" "$TXN" "$IGNORE" -m $ADDR
/Users/svn/bin/lint_check "$REPOS" "$TXN" "$IGNORE" -m $ADDR

# All checks passed, so allow the commit.
exit 0
```

For testing purposes, the checks can be run against known revisions using the -r flag. I frequently ran the whitespace check against revision 29 on my system when I wanted a successful test:

```
$ whitespace_check -r 29 /usr/local/svn/repos '[^\/]+/(?!thirdparty)/.+'
$ echo $?
```

```
0
```

Buildbot and Coverage

When I introduced Buildbot, I showed how to use it in conjunction with Nose to run unit tests. However, Nose can do much more. Together with the coverage package, it generates coverage reports. Coverage works with `easy_install` . . . sort of. At the moment (version 2.78), it is broken ever so slightly. The problem is known, and there is a patch for it.

All packages in the build we constructed are stored locally. This means that a locally patched version of coverage can be used. Retrieving the package from its distribution site is the first step. This is located by consulting `http://pypi.python.org`, the Python package repository.

The package is pulled down, unpacked locally, checked into source control, and subsequently patched. The version number is changed from 2.78 to 2.78p1 to designate that it has been patched.

A new package is generated with `python ./setup.py sdist`, and the resulting package file `dist/coverage-2.78p1.tar.gz` is copied into the project's `thirdparty` directory. The dependency on `coverage-2.78p1` is added to the project's `requires` attribute in `setup.py`.

The project is installed locally with `python ./setup.py install`, and the coverage tests are successfully run through Nose. This is done simply by adding the `--with-coverage` option when calling `nosetests`. The changes to the package are now known to function, so the patched files and the new third-party bundle are committed to their respective repositories.

Running coverage through Nose against an early version of svnhooks produces the following report:

```
$ nosetests -w src/test --with-coverage
```

```
...............................
Name                 Stmts   Exec  Cover   Missing
---------------------------------------------------
decorator               40     32    80%   62, 76-83, 123
getopt                 103     15    14%   43-45, 48, 79-93, 110-143, 146-162,
168-186, 189-201, 204-207, 210-211
pickle                 854    262    30%   84, 95-96, 197-207, 218, 222-225,
242-247, 251-257, 261-267, 271-331, 335, 339-343, 350-420, 427, 431-434, 438-458,
...
1358-1359, 1362, 1365-1367, 1370, 1373-1374, 1379-1380, 1383
pymock                  73     52    71%   31, 47-48, 66-67, 74-75, 80-81, 86-87,
92, 95, 98, 101, 106, 109, 112, 117, 122, 127
pymock.pymock          493    329    66%   78-81, 92-93, 101-103, 106, 116-123, 127,
144, 150, 153-154, 159, 162, 165-166, 171, 176, 181-182, 185-188, 191-193, 206,
...
734-735, 739-743, 747-750, 754-755, 759-760, 781, 789, 797, 806, 818, 824, 835-836
sets                   286     67    23%   60-79, 93-94, 101, 108, 114-117, 124,
132, 150-153, 156-159, 165-167, 178-185, 201-203, 210-212, 219-221, 228-235,
...
493-495, 499-505, 511, 515, 524-530, 537-543, 550-553, 557, 561, 565, 573-574, 577
subprocess             496     42     8%   368-369, 371, 375-398, 410, 415-417,
422-429, 443, 456-462, 493-530, 540-622, 626-628, 632-639, 653-667, 674-909,
...
```

```
1083-1089, 1095-1103, 1109-1112, 1116-1181, 1188-1222, 1229-1239, 1243-1246
svnhooks                  0      0   100%
svnhooks.indent          46     23    50%   17, 23-24, 28-31, 39-46, 49-51, 54-60
svnhooks.precommit       78     70    89%   32-33, 47-48, 51-52, 90, 99
svnhooks.syntax          18     17    94%   32
tabnanny                173     29    16%   28, 36-40, 44-58, 66, 68, 70, 72, 84-130,
156-176, 181-182, 199-203, 208, 215-223, 239-249, 256-264, 267-271, 274-325, 329
term                      0      0   100%
term.framework          814      0     0%   3-1123
--------------------------------------------------
TOTAL                  3474    938    27%
----------------------------------------------------------------------
Ran 31 tests in 0.089s

OK
```

The first column is the name of the package. The second is the number of statements in the file, followed by the number executed, and then the percentage calculated from those two. The final column is a list of lines and line ranges that were not covered.

The coverage report has one noticeable weak point. It doesn't distinguish between built-in packages and subject packages. The report simply includes all the modules that are imported.

The code coverage report is easily patched into Buildbot. The necessary changes to the configuration created in Chapter 5 are shown here in bold:

```
def python_(version):
   return  "../../python%s/bin/python" % version

def nosetests_(version):
   return "../../python%s/bin/nosetests" % version

def site_bin_(version):
   return  "../../python%s/site-bin" % version

def site_pkgs_(version):
    subst = {'v': version}
    path = "../../python%(v)s/lib/python%(v)s/site-packages"
    return path % subst

def pythonBuilder(version):
    python = python_(version)
    nosetests = nosetests_(version)
    site_bin = site_bin_(version)
    site_pkgs = site_pkgs_(version)
```

```
f = factory.BuildFactory()
f.addStep(SVN, baseURL="svn://repos/rsreader/",
   defaultBranch="trunk",
   mode="clobber",
   timeout=3600)
f.addStep(ShellCommand,
          command=["rm", "-rf", site_pkgs],
          description="removing old site-packages",
          descriptionDone="site-packages removed")
f.addStep(ShellCommand,
          command=["mkdir", site_pkgs],
          description="creating new site-packages",
          descriptionDone="site-packages created")
f.addStep(ShellCommand,
          command=["rm", "-rf", site_bin],
          description="removing old site-bin",
          description="site-bin removed")
f.addStep(ShellCommand,
          command=["mkdir", site_bin],
          description="creating new site-bin",
          descriptionDone="site-bin created")
f.addStep(ShellCommand,
          command=[python, "./setup.py", "setopt",
                     "--command", "easy_install",
                     "--option", "allow-hosts",
                     "--set-value", "None"],
          description="Setting allow-hosts to None",
          descriptionDone="Allow-hosts set to None")
f.addStep(Compile, command=[python, "./setup.py", "build"])
f.addStep(ShellCommand,
          command=[python, "./setup.py",  "install",
                   "--install-scripts", site_bin],
          description="Installing",
          descriptionDone="Installed")
f.addStep(ShellCommand,
          command=[python, "./setup.py", "test"],
          description="Running unit tests",
          descriptionDone="Unit tests run")
f.addStep(ShellCommand,
          command=[nosetests, "./src/test", "--with-coverage"],
          description="Determining code coverage",
          descriptionDone="Code coverage determined")
return f
```

Summary

People are error prone, and they're prone to inflated judgments about their own capabilities (of course, this excludes you and me). Programmers are people, and so they carry over the same faults. Without adequate feedback, they can't get an accurate assessment of their performance, so it is critical to get feedback one way or another.

Getting feedback translates to measuring and reporting some aspect of the development process. These measurements fall into two broad categories. Quantitative measurements answer the question, "How much?" They are the sorts of things that produce hard numbers, and are easily analyzed by computers. Qualitative measurements describe what something is, and they tend to be things that people are good at determining.

Measurements have many aspects, and they need to be considered before investing time and effort. You must understand the characteristics of what you are measuring, the characteristics of the instrument you are measuring with, and the interactions between the two. You must understand what the measurements mean, what the impact of measuring will be, and what the possible adverse outcomes are.

Quantitative measurements tend to have very narrow definitions, and are only applicable in limited scopes. Many quantitative measurements are required to get an accurate assessment of a software project's overall condition. Among those commonly encountered are code coverage, cyclomatic complexity, and velocity.

Qualitative measurements tend to have much broader application, and they give a much deeper picture of a project's condition. They're also much harder to determine since they require human intervention, and collecting them systematically often requires procedural support. The granddaddy of all qualitative measurement regimes is the code review, although in agile development environments, pair programming often takes the place of formal reviews.

Such measurement regimes provide feedback about our behavior and work. Feedback can be achieved by social or technological means, but you should be wary of the temptation to use technology to solve social problems, as cultural norms and peer pressure are often more effective ways of shaping behavior.

Technological solutions have limited domains, but are often effective when used appropriately. There are a number of tools that help to provide feedback. Eclipse, Pydev, Subversion, and Buildbot can all provide useful feedback when used correctly.

Chapter 9 looks at databases. Databases are their own little world, and they present thorny issues when it comes to agile development. The issues of incremental upgrades and downgrades to live databases are largely solved, but only in ad hoc ways. Moreover, these mandate very different machinery from the object-relational mappers that are used to access databases today.

CHAPTER 9

Databases

Databases have traditionally been treated as entities distinct from the larger code base. This is reflected organizationally; a firm wall often exists between developers and database administrators (DBAs); the DBAs work in parallel with the rest of the organization.

This is rooted in the historical nature of the technology. In the past, databases have been expensive both in terms of software and hardware, and this means that there haven't been very many of them. With so few resources, very few people acquired the skills to work with them. Dedicated staff were required to gate access to these limited resources, and to prevent the naive from doing stupid things. Additionally, most products were very hard to configure. Getting acceptable performance required much tuning, and thus even more expertise. The company jewels were often stored in these mines, and had to be protected from fumbling hands and untested scripts.

In recent years, the technical landscape has changed. Over the last ten years, free database implementations have blossomed, as have computing and storage capabilities.

This has given rise to a proliferation of databases. SQL databases have morphed from beasts with complicated interacting processes and dedicated raw filesystem drivers to serverless libraries that can be linked and embedded within shipping code. Examples of these include HSQLDB, embedded MySQL, and SQLite. As of Python 2.5, SQLite even ships in the standard library.

Every developer can now have a database on the desktop (or laptop or palmtop). The consequences of this have been slow to sink in. Agile software development techniques have a long history behind them, but agile database development techniques do not.

A New Religion

The ultimate goal of any development organization is delivering business value. While the proximate goal of development is producing software and the proximate goal of a database is organizing data for retrieval, these are not the ultimate organizational goals. If the CEO could get the same information more cheaply and reliably by calling a televangelist, then she'd be doing it.

Neither the goals of development nor the goals of the DBA organization are meaningful without assistance from the each other. Development wants to use the databases to accomplish meaningful work for the company, and the DBAs want to ensure that the company's data is protected.[1] These two organizations are often at loggerheads when they should be working in concert to meet the overall organization's needs.

Agile development recognizes that different business groups have differing needs and priorities, and that these change over time. Code must change to reflect these realities, and this leads to the need for constant refactoring. The same is true of data models. Like Code, they will rot if they're not regularly maintained.

The database groups need to work closely with their customers to understand these issues. As with development, they need to focus on the issues with the biggest payoffs. The work should be prioritized, and some things will fall by the wayside. There will always be new problems, so the database organization shouldn't try to solve everything. The key is not to try to eliminate and address all problems, but to design a process that addresses new issues as normal occurrences.

Blurring the Boundaries

Only by creating integrated and fully automated processes can an organization meet the rapid turnaround required by short iterations, and this can only be done by integrating the automation into the entire production cycle from start to finish. Agile development breaks down some of the separations between development, operation, and administration. Agile development therefore has strong impacts on the DBA organization:

Database design becomes an *evolutionary process*. Since change is a constant pressure, the database schema is never complete. These changes must be propagated quickly from development through to production. This must be done in such a way that it can be replicated, and it must be done without human intervention.

Databases are improved through *refactorings*. These are changes that improve the structure of the database without altering its function. The need to accommodate live changes imposes certain design constraints not present in code.

Code must be *isolated* from the underlying data model as much as possible. Much is written about an object-relational mismatch. I don't subscribe to that view any more than I subscribe to a view of an object-filesystem mismatch or an object-thread mismatch. Relational databases are complicated, but that doesn't mean that there is a fundamental misfit. It does mean that there is a lot of machinery required to magically unify the two.

Testing must be performed. Changes must be made to the database. Changes must also be made to the code that uses the database. A variety of techniques are used to accomplish these tests. Some require little more than the machinery already discussed in previous chapters, and some require new classes of software.

1. The DBAs should ensure that the company's data is available and protect it from loss. Often the first is forgotten, but if nobody is doing useful work with the production databases, then either the databases are superfluous or the company is in dire trouble. Either way, the DBAs are in trouble.

Developers and DBAs both have a role in this, but since many tools reside in the software development process, the DBAs have to learn more about those tools and processes. At the same time, developers will have to learn more about being a DBA. The DBA's job becomes less about adjudicating changes and more about providing expertise and advising against absolute stupidity. Because there is no clear organizational boundary, the DBAs have to work closely with the developers to ensure that proper procedural boundaries are observed.

Concealing Data Access

At some point, your code has to talk to the database. At that point, the code needs to understand the details of the data. It must know how to locate the data source and initiate a conversation. It must know the structure of the data to perform efficient queries. It needs to convert between local types and stored types, and back again, and it must know how and when to write out changes. It must be able to recognize stale results, and it often needs to cache data that is expensive to retrieve from the database.

When the structure of the data changes, the code that accesses that data needs to change. If the data access code is scattered throughout a program, then every change necessitates seeking those points out and rewriting the access code. This is time-consuming and prone to error.

Therefore, code dealing with the database should be in a central location. This layer mediates all access to the database. It can be as simple or as complex as needed. At one end of the spectrum, it might simply be a few methods that read and write strings to a file. At the other end are systems that map between relational databases and classes or objects within a program.

Such libraries are called object-relational mappers (ORMs). These subsystems provide an elaborate framework concealing the details of the underlying query mechanisms. They make it easy to interface with the underlying database systems. With a good ORM, it is easier to write database access code than it is to work with files.

Object-Relational Mappers

ORMs generally have four aspects:

- A description of the database schema
- A mapping between the schema and the application objects
- A way of selecting data
- A mechanism for writing changes

ORMs differ widely in how these are aspects are handled. In some cases, they are manually specified. In others, they are automatically derived from a running system. In some cases, the running system's configuration is derived from the ORM definitions.

I'm going to discuss the two leading Python ORMs: SQLObject and SQLAlchemy. There are three common patterns that are useful when discussing them:

- Active record
- Data mapper
- Unit of work

The Active Record Pattern

The active record pattern describes a simple relationship between a database and the programming language. A database table corresponds to a class, a row in a table corresponds to an instance of the class, and a column corresponds to an attribute.

Queries return objects, and the values are read from the attributes. Writing to an attribute updates the database. Creating an instance inserts a row. Deleting an object deletes the row. Inherent in the active record pattern is the idea that each row has an identity.

This pattern is easy to describe and understand. It combines the steps of describing the database schema and producing a mapping between the schema and application objects. It has the advantage of working very well for small-to-medium-sized cases.

While it easily maps tables, rows, and columns, it doesn't easily map other database objects, such as procedure results, views, joins, column selects, and multitable or multidatabase results.

The biggest problem with the active record pattern is that the resulting code closely mirrors the database schema. When the database structure changes, the code must also change, and these changes are distributed throughout the code. Solving this requires a layer of indirection.

The Data Mapper Pattern

The data mapper pattern maps columns into arbitrary objects. The underlying structure is described, and then the mappings are specified between the storage entities and the application objects.

This indirection separates the database from the application. The storage format can be altered, while the objects remain the same, and vice versa. Changing the database structure no longer necessitates changing the application code, and arbitrary SQL results can be sensibly mapped.

On the other hand, it's a little more complicated to set up. It hides database access and structure by distributing them throughout your code. The relationships between attributes in one place and those in another can be concealed. It's a little harder to understand what is going on in some cases.

The Unit of Work Pattern

In this pattern, the code tracks the changes that have been made and commits them in a single batch within a single transaction.

Talking to the database is expensive. Each batch of changes incurs a significant time lag. Often the majority of an application's time is spent waiting for results from the database. I have personally seen situations in which more than 90 percent of an application's response time was spent waiting on the database. The actual code took microseconds to run, but each

round trip to the database took milliseconds. Committing the changes in a single batch reduces this overhead dramatically.

Since the database transaction is only held for the length of the batched connection, there is less contention between queries and less opportunity for deadlock. The application quickly uses and returns connections, so the running application needs to have fewer open connections to the database in order to achieve the same throughput.

The application is in control of the commits, so it knows when problems occur. The commit points also provide a natural point to handle rollback.

There are disadvantages, though. Control comes at the expense of effort and forethought. Developers must be aware of when changes are committed and how the batches are constructed. Potentially, an application can continue running with uncommitted changes that haven't been rolled back, leading to inconsistent views of the database and possible loss of data. The application may be less responsive. While its overall performance may increase, the lower latency of a do-it-immediately approach may be worth the increase in responsiveness. A straight do-it-now access policy is useful and appropriate for many small applications.

Python ORMs

There are many Python ORMs, but there are two 900-pound gorillas. They are SQLObject and SQLAlchemy. SQLObject has been around quite a bit longer than SQLAlchemy, but the latter is gaining in popularity. Although more complicated for novices, it is far more capable when it comes to real production problems.

SQLObject

SQLObject is based on the active record pattern. It has minimal support for the unit of work pattern, and many people simply write to the database. It has an aggressive caching policy by default, and it uses a simple declarative format to specify both the schema and mappings. It really wants to use numeric keys for database records.

As always, obtaining the package is the first step:

```
$ easy_install -U SQLObject
```

```
Searching for SQLObject
Reading http://pypi.python.org/simple/SQLObject/
...
Processing dependencies for SQLObject
Finished processing dependencies for SQLObject
```

I'm using a classic example—that of students in a school. The `student` table looks like Figure 9-1.

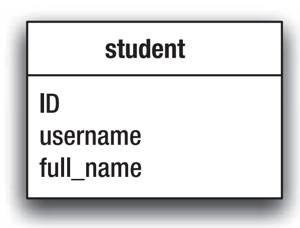


Figure 9-1. *The student table*

The schema for this table might be generated by the following SQL:

```
CREATE TABLE student (
    ID INTEGER PRIMARY KEY AUTOINCREMENT,
    full_name VARCHAR(64) NOT NULL,
    username VARCHAR(16) NOT NULL
);
```

This table would be described to SQLObject as follows:

```
from sqlobject import SQLObject, StringCol

class Student(SQLObject):
    username = StringCol(length=16)
    fullName = StringCol(length=64)
```

Connecting to the Database

The next step is establishing a connection to the database. SQLObject uses standard connection URI syntax:

```
scheme://[user[:password]@]host[:port]/database[?parameters]
```

Examples include the following:

- `mysql://jeff:myPasswordHere@localhost/test_db`
- `postgres://bob@my.host.com/another_db?debug=1&cache=0`
- `postges:///path/to/socket/db_name`
- `sqlite:///path/to/the/database`

As of version 0.9, the common parameters are as follows:

- `debug`
- `debugOutput`
- `debugThreading`
- `cache`

- `autoCommit`
- `logger`
- `logLevel`

Since SQLite ships with Python, I'll be using it for the examples. The following code fragment sets up a SQLite connection:

```
filename = "test_db"
abs_path = os.path.abspath(filename)
connection_uri = 'sqlite://' + abs_path
connection = sqlobject.connectionForURI(connection_uri)
sqlobject.sqlhub.processConnection = connection
```

You can turn this into the following method:

```
def sqlite_connect(abs_path):
    connection_uri = 'sqlite://' + abs_path
    connection = sqlobject.connectionForURI(connection_uri)
    sqlobject.sqlhub.processConnection = connection
```

The important thing is that you set the `processConnection` variable to the correct connection. If you turn this into a method, the corresponding test is as follows:

```
@use_pymock
def test_sqlite_connect():
    f = '/x'
    uri = 'sqlite:///x'
    connection = dummy()
    override(sqlobject, 'connectionForURI').expects(uri).\
        returns(connection)
    replay()
    sqlite_connect(f)
    assert sqlobject.sqlhub.processConnection is connection
    verify()
```

Creating Rows

New rows are created by instantiating objects. Here's a simple test for this:

```
s1 = Student(username="jeff", fullName="Jeff Younker")
assert s1.username == "jeff"
assert s1.fullName == "Jeff Younker"
```

There's a good deal of setup and tear-down that needs to be done, though. A new database file must be created, and the connection to that database must be initiated. At the end of the test, the file should be removed, the object cache should be cleared to prevent other tests from stomping on yours, and finally the connection should be closed.

Note The connection hub's caching plays havoc with the SQLite driver, so the test generates a new randomly named connection each time.

```
import random
...
def random_string(length):
    seq = [chr(x) for x in range(ord('a'), ord('z')+1)]
    return ''.join([x for x in random.sample(seq, length)])

def test_creating_student():
    f = os.path.abspath(random_string(8) + '.db')
    if os.path.exists(f):
        os.unlink(f)
    sqlite_connect(f)
    try:
        s1 = Student(username="jeff", fullName="Jeff Younker")
        assert s1.username == "jeff"
        assert s1.fullName == "Jeff Younker"
    finally:
        sqlobject.sqlhub.processConnection.cache.clear()
        sqlobject.sqlhub.processConnection.close()
        del sqlobject.sqlhub.processConnection
        os.unlink(f)
```

When this runs, it gives the following error:

```
Traceback (most recent call last):
  File "/Library/Python/2.5/site-packages/nose-0.10.0-py2.5.egg/nose/case.py",➥
 line 202, in runTest
    self.test(*self.arg)
...
  File "/Users/jeff/Library/Python/2.5/site-packages/SQLObject-0.10.0b2-py2.5.egg/➥
sqlobject/sqlite/sqliteconnection.py", line 177, in _executeRetry
    raise OperationalError(ErrorMessage(e))
OperationalError: no such table: student
```

In other words, the schema has not been defined yet. The tests could create the schema directly, but that ties them to the specific database used for the unit tests. Fortunately, `SQLObject` instances know how to create themselves. One command creates this new table. The revised test method is as follows:

```
def test_creating_student():
    f = os.path.abspath('test_db')
    if os.path.exists(f):
        os.unlink(f)
```

```
    sqlite_connect(f)
    try:
        Student.createTable()
        s1 = Student(username="jeff", fullName="Jeff Younker")
        assert s1.username == "jeff"
        assert s1.fullName == "Jeff Younker"
    finally:
        sqlobject.sqlhub.processConnection.cache.clear()
        sqlobject.sqlhub.processConnection.close()
        del sqlobject.sqlhub.processConnection
        os.unlink(f)
```

The test now runs successfully to conclusion. It's a mess, though, and there are going to be many more of these written. The setup and tear-down can be refactored into a decorator:

```
from decorator import decorator
...
@decorator
def with_sqlobject(tst):
    f = os.path.abspath(random_string(8) + '.db')
    if os.path.exists(f):
        os.unlink(f)
    sqlite_connect(f)
    try:
        Student.createTable()
        tst()
    finally:
        sqlobject.sqlhub.processConnection.cache.clear()
        sqlobject.sqlhub.processConnection.close()
        os.unlink(f)

@with_sqlobject
def test_writing_student():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    assert s1.username == "jeff"
    assert s1.fullName == "Jeff Younker"
```

The resulting test is significantly more concise. The preceding code uses the decorator module, which is a third-party module that simplifies writing decorators. Most decorators usually involve creating at least one closure, and this closure is nearly always the same. Here's a decorator that prints `before` and then executes the wrapped function:

```
def before(f):
    def wrapper(*args, *kw):
         print "before"
         return f(*args, **kw)
    return wrapper
```

The decorator module supplies the necessary closure machinery:

```
from decorator import decorator
...
@decorator
def before(f, *args, **kw)
    print "before"
    return f(*args, **kw)
```

I find the resulting decorators much cleaner and easier to understand.

Putting the Schema Where It Belongs

Right now there is only one table, but eventually there will be many. Every time a new table is added, the schema definition in `with_sqlobject()` will grow. This schema creation information may also be useful in the program itself, particularly when it needs to be installed, so it should go into the file with the schema declarations.

```
from sqlobject_ex import create_schema
...
@decorator
def with_sqlobject(tst):
    f = os.path.abspath(random_string(8) + '.db')
    if os.path.exists(f):
        os.unlink(f)
    sqlite_connect(f)
    try:
        create_schema()
        tst()
    finally:
        sqlobject.sqlhub.processConnection.cache.clear()
        sqlobject.sqlhub.processConnection.close()
        os.unlink(f)
```

And the `create_schema()` method should go into `sqlobject_ex.py`:

```
def create_schema():
    Student.createTable()
```

Attribute Defaults

What happens if one of the student attributes is omitted? For example

```
>>> Student(fullName="Jeff Younker")
```

gives the following error:

```
Traceback (most recent call last):
  ...
ValueError: Unknown SQL builtin type: <type 'classobj'> for <class sqlobject.sql➥
builder.NoDefault at 0xde05d0>
```

All attributes are required unless a default is defined. In other words, all attributes are assumed to be `NOT NULL` unless declared otherwise with the `default` attribute. The following code makes the username optional:

```
class Student(SQLObject):
    username = StringCol(length=16, default=None)
    fullName = StringCol(length=64)
```

Selecting Objects

SQLObject has three methods for retrieving objects from the database. The `get()` method retrieves a single object by its ID. The attribute `id` maps to the field ID. It is transparently managed by the ORM. All mapped tables must have an ID field.

```
@with_sqlobject
def test_get():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student.get(s1.id)
    assert s1 is s2
```

The `select()` class method chooses one or more objects. With no arguments, it returns all instances in the table.

```
from sets import Set
...
@with_sqlobject
def test_select():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    students = list(Student.select())
    assert len(students) == 2
    assert Set(students) == Set([s1, s2])
```

The `select()` method takes a SQLBuilder expression. SQLBuilder is part of SQLObject. You build SQL queries from SQLBuilder calls. The package makes extensive use of operator overloading, so for simple cases, queries look just like normal Python comparison expressions.

```
@with_sqlobject
def test_select_using_full_name():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    unused_s2 = Student(username="doug", fullName="Doug McBride")
    students = Student.select(Student.q.fullName == "Jeff Younker")
    assert list(students) == [s1]
```

The class variable `Student.q` contains column descriptions. These are used in SQLBuilder queries. The preceding expression translates to the following SQL:

```
select * from student where full_name = "Jeff Younker"
```

For simple comparisons, you'll never have to access SQLBuilder directly, but more esoteric expressions require more direct meddling. The following code uses a SQL-like expression to search for all students with a full name containing *ou*.

```
from sqlobject.sqlbuilder import LIKE
...
@with_sqlobject
def test_select_using_partial_name():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    unused_s3 = Student(username="amy", fullName="Amy Woodward")
    students = Student.select(LIKE(Student.q.fullName, '%ou%'))
    assert Set(students) == Set([s1, s2])
```

The `selectBy()` method is a concise method of querying exact column matches. Keywords specify the attributes to be compared, and the values are those to be compared with.

```
@with_sqlobject
def test_selectBy_full_name():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    unused_s3 = Student(username="amy", fullName="Amy Woodward")
    students = Student.selectBy(fullName="Jeff Younker")
    assert list(students) == [s1]
```

Like `select()`, if no arguments are supplied, it returns the entire table:

```
@with_sqlobject
def test_selectBy_all():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    students = Student.selectBy()
    assert Set(students) == Set([s1, s2])
```

Updating Fields

Values are modified via simple assignment:

```
@with_sqlobject
def test_modify_values():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s1.fullName = "Jeff M. Younker"
    students = Student.selectBy(fullName="Jeff M. Younker")
    assert list(students) == [s1]
```

Deleting Rows

All `SQLObject` instances have a `destroySelf()` method. Calling this method deletes the associated row from the database:

```
@with_sqlobject
def test_delete():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s1.destroySelf()
    students = Student.select()
    assert list(students) == []
```

The `destroySelf()` method does not perform cascading deletes, but that can be accomplished by overriding this method.

One-to-Many Relationships

SQLObject specifies joins (specifically inner joins) declaratively. The products of the joins appear as arrays contained in instance variables. To demonstrate, I've expanded the schema to include an e-mail address for each student. Each student may have more than one e-mail address (see Figure 9-2).

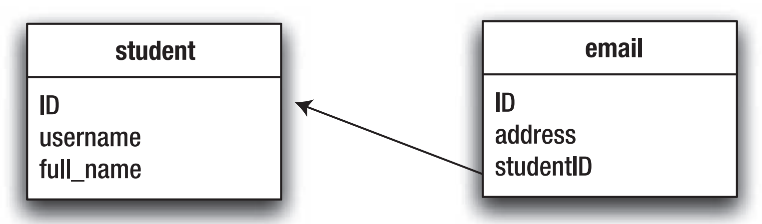


Figure 9-2. *A many-to-one relationship between student and e-mail address*

The corresponding SQL for the new table is as follows:

```
CREATE TABLE email (
    ID INTEGER PRIMARY KEY AUTOINCREMENT,
    address VARCHAR(255) NOT NULL,
    studentID INTEGER NOT NULL,
    FOREIGN KEY studentID REFERENCES student(id)
);
```

Foreign Keys

The new table is defined in `sqlobject_ex.py` as follows:

```
from sqlobject import ForeignKey, SQLObject, StringCol
...
class Email(SQLObject):
    email = StringCol(length=255)
```

```
    student = ForeignKey('Student')
...
def create_schema():
    Student.createTable()
    Email.createTable()
```

ForeignKey defines the attribute student as a link to the class Student. When this attribute is accessed, the key studentID will be dereferenced and the row will be instantiated. The IDs are handled under the hood.

```
@with_sqlobject
def test_email_creation():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    e1 = Email(address="jeff@not.real.org", student=s1)
    assert e1.student is s1
    e1.student = s2
    assert e1.student is s2
```

SQLObject foreign keys are always expected to end in ID. This is one of the drawbacks of using SQLObject. The underlying foreign key can be accessed directly via the attribute:

```
@with_sqlobject
def test_direct_id_access():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    e1 = Email(address="jeff@not.real.org", student=s1)
    assert e1.studentID == s1.id
    e1.studentID = s2.id
    assert e1.student is s2
```

Multiple Joins

So far, if the code has an Email, it can locate the associated Student, but there is no way to go in the other direction. The MultipleJoin class provides this functionality. You modify the Student class like this:

```
from sqlobject import ForeignKey, MultipleJoin, SQLObject, StringCol
...
class Student(SQLObject):
    fullName = StringCol(length=64)
    username = StringCol(length=16)
    emails = MultipleJoin('Email')
```

Accessing the emails attribute returns a list of associated Email objects:

```
@with_sqlobject
def test_multiple_join():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    e1 = Email(address="jeff@not.real.org", student=s1)
    assert s1.emails == [e1]
```

If there are no objects, then an empty list is returned:

```
@with_sqlobject
def test_multiple_join_empty_returns_empty_list():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    assert s1.emails == []
```

The attribute looks like it should be mutable, but you can't assign to it:

```
@with_sqlobject
def test_multiple_join_cant_assign():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    e1 = Email(address="jeff@not.real.org", student=s1)
    try:
        s1.emails = [e1]
        assert False
    except AttributeError:
        pass
```

`MultipleJoin` attributes are read-only. The only way to alter their contents is by changing the foreign key:

```
@with_sqlobject
def test_changing_a_multiple_join():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    s2 = Student(username="doug", fullName="Doug McBride")
    e1 = Email(address="jeff@not.real.org", student=s1)
    e2 = Email(address="doug@not.real.org", student=s2)
    assert s1.emails == [e1]
    e2.student = s1
    assert Set(s1.emails) == Set([e1, e2])
```

Many-to-Many Relationships

Students are in many classes, and classes contain many students. This kind of relationship is referred to as a many-to-many relationship. In relational databases, these are expressed through intermediate tables. Each entry is essentially a double-ended pointer to the tables it relates (see Figure 9-3).

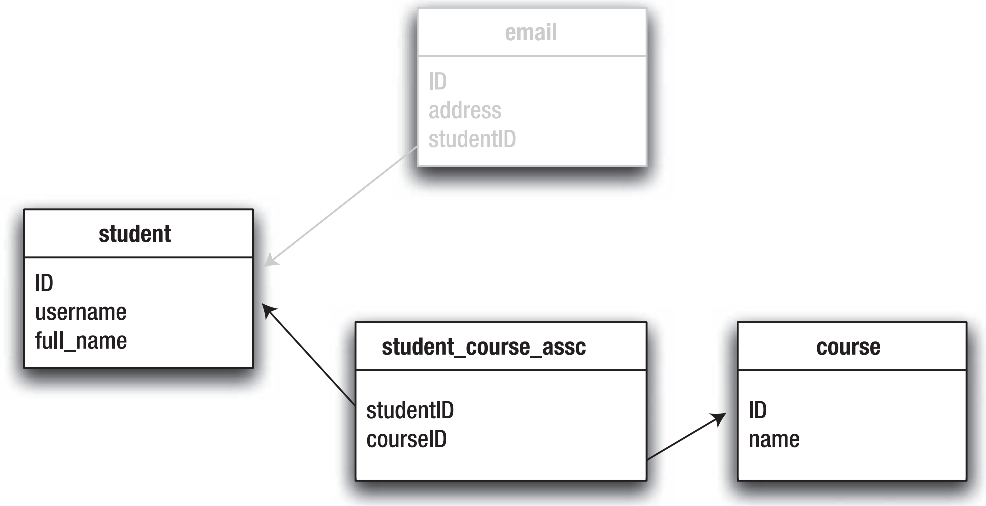


Figure 9-3. *A many-to-many relationship between students and classes*

The SQL defining these tables in SQLite is as follows:

```
CREATE TABLE course (
    ID INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(64) NOT NULL
);

CREATE TABLE student_course_assc (
    studentID INTEGER NOT NULL,
    courseID INTEGER NOT NULL,
    FOREIGN KEY studentID REFERENCES student(ID),
    FOREIGN KEY courseID REFERENCES course(ID)
);
```

Only the target class is defined. The intermediate table is implicit in the `RelatedJoin` declarations. The components related to the join are shown in bold:

```
from sqlobject import ForeignKey, MultipleJoin, RelatedJoin, \
    SQLObject, RelatedJoin, StringCol
...
def create_schema():
    Student.createTable()
    Email.createTable()
    Course.createTable()
...
```

```
class Student(SQLObject):
    fullName = StringCol(length=64)
    username = StringCol(length=16)
    emails = MultipleJoin('Email')
    courses = RelatedJoin('Course')
...
class Course(SQLObject):
    name = StringCol(length=64)
    students = RelatedJoin('Student')
```

Joining Students and Courses

The join statements create add and remove methods. These are named after the class being joined. In the `Student` class, they would be `addCourse()` and `removeCourse()`. As with a multiple join, the attribute returns a list of associated objects:

```
from sqlobject_ex import Course, Email, sqlite_connect, Student, create_schema
...
@with_sqlobject
def test_related_join_add():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    c2 = Course(name="Biochemistry")
    s1.addCourse(c1)
    s1.addCourse(c2)
    assert Set(s1.courses) == Set([c1, c2])
    assert c1.students == [s1]
    assert c2.students == [s1]
```

The relationship can be established from either end:

```
@with_sqlobject
def test_related_join_add_in_other_order():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    c2 = Course(name="Biochemistry")
    s1.addCourse(c1)
    c2.addStudent(s1)
    assert Set(s1.courses) == Set([c1, c2])
    assert c1.students == [s1]
    assert c2.students == [s1]
```

Relations are removed with the `remove`*`Foo`*`()` method:

```
@with_sqlobject
def test_related_join_remove():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    c2 = Course(name="Biochemistry")
    s1.addCourse(c1)
    s1.addCourse(c2)
    assert Set(s1.courses) == Set([c1, c2])
    c2.removeStudent(s1)
    s1.removeCourse(c1)
    assert s1.courses == []
    assert c1.students == [] and c2.students == []
```

Adds can be performed multiple times, and they result in multiple records:

```
@with_sqlobject
def test_related_join_multiple_adds():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    s1.addCourse(c1)
    s1.addCourse(c1)
    assert s1.courses == [c1, c1]
    assert c1.students == [s1, s1]
```

Removes take away all the duplicates:

```
@with_sqlobject
def test_related_join_removing_multiples():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    s1.addCourse(c1)
    s1.addCourse(c1)
    s1.removeCourse(c1)
    assert s1.courses == []
```

Multiple Relationships

Multiple relationships are frequently created between two tables. For example, a student may be enrolled in a course or have completed a course. A corresponding schema is shown in Figure 9-4.

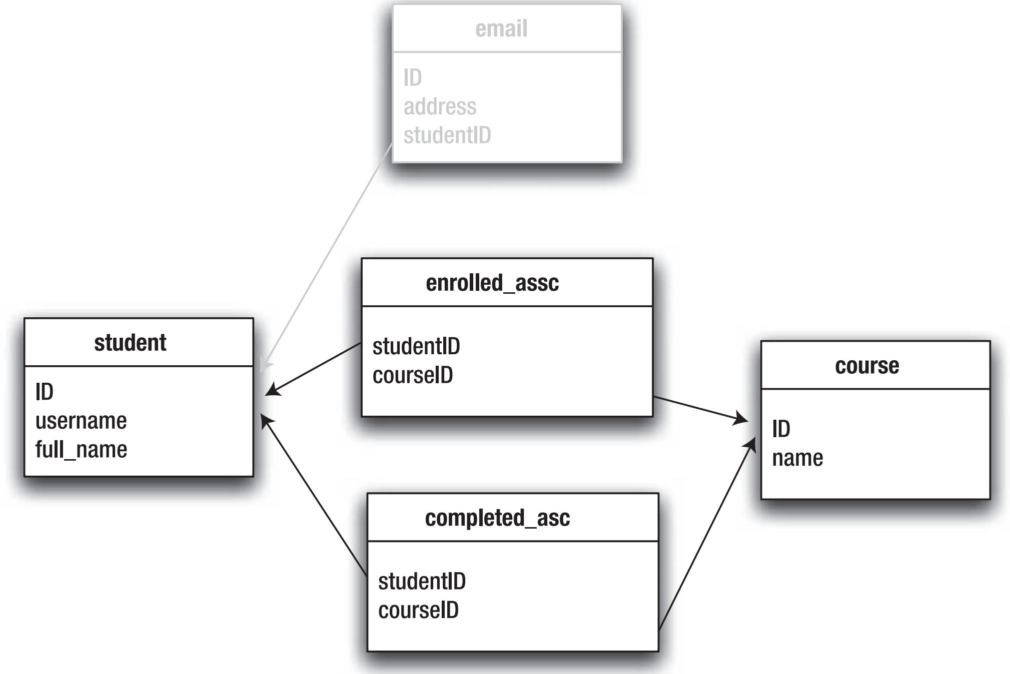


Figure 9-4. *A student can be enrolled in a course or may have completed a course.*

The SQLObject model is modified to reflect this:

```
class Student(SQLObject):
    fullName = StringCol(length=64)
    username = StringCol(length=16)
    emails = MultipleJoin('Email')
    enrolled = RelatedJoin('Course',
                           intermediateTable="enrolled_assc",
                           joinColumn="studentID",
                           otherColumn="courseID",
                           addRemoveName="Enrolled")
    completed = RelatedJoin('Course',
                            intermediateTable="completed_assc",
                            joinColumn="studentID",
                            otherColumn="courseID",
                            addRemoveName="Completed")

class Email(SQLObject):
    address = StringCol(length=255)
    student = ForeignKey('Student')
```

```
class Course(SQLObject):
    name = StringCol(length=64)
    enrolled = RelatedJoin('Student',
                           intermediateTable="enrolled_assc",
                           joinColumn="courseID",
                           otherColumn="studentID",
                           addRemoveName="Enrolled")
    completed = RelatedJoin('Student',
                            intermediateTable="completed_assc",
                            joinColumn="courseID",
                            otherColumn="studentID",
                            addRemoveName="Completed")
```

As with the simple `RelatedJoin`, the first argument is the class being joined with the containing class. The table containing the relation is specified with the `intermediateTable` keyword. The `joinColumn` and `otherColumn` keywords specify column names in the relation table. The join column points back to the containing class, and the other column links to the contained class. If left to its own devices, SQLObject generates the add and remove methods from the class name. The `addRemoveName` keyword supplies a different one.

■ **Note** It is not necessary for these definitions to be symmetrical. If your application will only add courses to students, and you can guarantee that courses will never be queried for the students they contain, then the related join can be omitted from the course.

At this point, all of the tests relating `Course` instances to `Student` instances have become invalid, and they need to be removed. Running the test suite indicates which ones should be removed.

The new `enrolled` relation and its add and remove methods are shown in this test:

```
@with_sqlobject
def test_enrollment_add_and_remove():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    s1.addEnrolled(c1)
    assert s1.enrolled == [c1]
    s1.removeEnrolled(c1)
    assert s1.enrolled == []
```

And it is clear that the two new relations point to separate tables:

```
@with_sqlobject
def test_enrollment_relations_are_separate():
    s1 = Student(username="jeff", fullName="Jeff Younker")
    c1 = Course(name="Modern Algebra")
    c2 = Course(name="Biochemistry")
    s1.addEnrolled(c1)
```

```
    s1.addCompleted(c2)
    assert s1.enrolled == [c1]
    assert s1.completed == [c2]
```

SQLAlchemy

SQLObject focuses on making easy things easy, but hard things are still hard. SQLAlchemy requires more configuration, but it pays that back in power. The results of any arbitrary join or select statement may be mapped to objects. The package gives fine-grained control over object graph loading and saving. It provides connection pooling and a low-level database interface. Generated SQL can even be replaced with custom queries when needed.

It is obtained via `easy_install`:

```
$ easy_install SQLAlchemy
```

```
Searching for SQLAlchemy
Reading http://pypi.python.org/simple/SQLAlchemy/
...
Installed /Users/jeff/Library/Python/2.5/site-packages/SQLAlchemy-0.4.3-py2.5.egg
Processing dependencies for SQLAlchemy
Finished processing dependencies for SQLAlchemy
```

As with SQLObject, the examples here will use SQLite. Without SQLObject's caching issues, an in-memory database can be used safely. The application code is defined in `sqlalchemy_ex.py`, and the tests are defined in `test_sqlalchemy_ex.py`.

With SQLAlchemy, setting up a connection is simple enough that it doesn't justify encapsulating the code:

```
from sqlalchemy import create_engine

def test_connection():
    engine = create_engine('sqlite:///:memory:')
```

The schema for the `student` table is pictured in Figure 9-5. The table is defined in `sqlachemy_ex.py`.

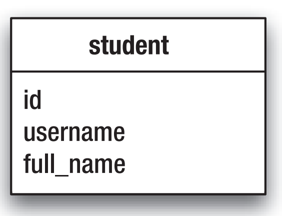


Figure 9-5. *The student table again*

```
from sqlalchemy import Column, Integer, MetaData, Table, String

schema = MetaData()
student_table = Table('student', schema,
                      Column('id', Integer, primary_key=True),
                      Column('username', String(16)),
                      Column('full_name', String(64)),
)
```

The MetaData class describes a connection. The Table() method describes the table's schema and associates it with the MetaData object (called schema). The declaration looks very much like a SQL table statement. Unlike SQLObject, the primary key's identity and type is not assumed, and must be declared.

This test creates the schema in the connected database:

```
from sqlalchemy_ex import schema
...
def test_schema_creation():
    engine = create_engine('sqlite:///:memory:')
    schema.create_all(engine)
```

This creates a schema, but there is no way to alter data yet. Doing this involves two steps. First, a class must be declared, and then the table must be mapped to the class. This linkage occurs through the names of table columns and object attributes.

```
from sqlalchemy import Column, Integer, MetaData, Table, String
from sqlalchemy.orm import mapper

schema = MetaData()
student_table = \
    Table('student', schema,
        Column('id', Integer, primary_key=True, nullable=False),
        Column('username', String(16), nullable=False),
        Column('full_name', String(64), nullable=False),
)

class Student(object):

    def __init__(self, username, full_name):
        self.username = username
        self.full_name = full_name

mapper(Student, student_table)
```

Notice that the primary key id is implicitly mapped. SQLAlchemy understands the significance of the primary key, and the mapper automatically manages it for you. SQLAlchemy draws a distinction between creating an object and saving it to the database. Until the object is saved, the id is None.

While SQLObject columns assume that columns are `IS NOT NULL`, SQLAlchemy columns assume the opposite. The `nullable` keyword allows the code to change this. In the table just defined, each column explicitly sets `nullable` to `False`.

```
def test_create_unsaved_student():
    s1 = Student(username="jeff", full_name="Jeff Younker")
    assert s1.username == "jeff"
    assert s1.full_name == "Jeff Younker"
    assert s1.id is None
```

Manipulating data requires a session. Sessions come from `Session` classes, which are in turn created by the `sessionmaker()` function. The session must be bound to a database engine at some point, which can be done either when it is created or afterward. The following tests demonstrate both methods:

```
from sqlalchemy.orm import sessionmaker
...
def test_getting_a_session():
    engine = create_engine('sqlite:///:memory:')
    schema.create_all(engine)
    Session = sessionmaker(bind=engine, autoflush=True,
       transactional=True)
    unused_session = Session()


def test_getting_a_session_and_binding_later():
    engine = create_engine('sqlite:///:memory:')
    schema.create_all(engine)
    Session = sessionmaker(autoflush=True, transactional=True)
    Session.configure(bind=engine)
    unused_session = Session()
```

Saving an object adds it to the session, but the session does not instantly flush the changes to the database. A flush can be manually forced:

```
def test_create_and_save_student():
    engine = create_engine('sqlite:///:memory:')
    schema.create_all(engine)
    Session = sessionmaker(bind=engine, autoflush=True, \
      transactional=True)
    session = Session()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    session.save(s1)
    assert s1.id is None
    session.flush()
    assert s1.id is not None
```

The test methods are getting a little unwieldy at this point, so refactoring the code is a good idea. The database and session configuration is refactored into a new method:

```
def session_from_new_db():
    engine = create_engine('sqlite:///:memory:')
    schema.create_all(engine)
    Session = sessionmaker(bind=engine, autoflush=True,
      transactional=True)
    return Session()

def test_create_and_save_and_flush_student():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    session.save(s1)
    assert s1.id is None
    session.flush()
    assert s1.id is not None
```

Flushing is required in the preceding code, but during normal operation, `autoflush` will trigger a flush at appropriate times.

The next test saves a student and then retrieves it:

```
def test_retrieve_from_database():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    session.save(s1)
    f = session.query(Student).filter_by(username="jeff").first()
    assert f is s1
    assert s1.id is not None
```

It is clear that a flush happened immediately before the query, since it found the new record, and because `id` has been set.

When working in transactional mode, it is necessary to commit the changes before they become permanent:

```
def test_commit_changes():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    session.save(s1)
    session.commit()
```

Committing the session flushes all saved changes in a single transaction, closes it, and begins a new one.

As with SQLObject, attributes map through to the underlying columns in the database, and they can be modified as if they were instance variables:

```
def test_set_and_modify_database():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    session.save(s1)
    #
    f = session.query(Student).filter_by(full_name=\
      "Jeff M. Younker").first()
```

```
    assert f is None
    #
    s1.full_name = "Jeff M. Younker"  # flush happens before query
    f = session.query(Student).filter_by(full_name=\
      "Jeff M. Younker").first()
    assert f is s1
```

Queries

All queries begin with the `query()` method; this has been show previously, but not noted. Queries differ in the subsequent filtering commands. With no filtering, a query returns all the rows in a table:

```
def test_query_all_rows():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    s2 = Student(username="doub", full_name="Doug McBride")
    session.save(s1)
    session.save(s2)
    f = session.query(Student)
    assert Set(f) == Set([s1, s2])
```

Choosing Results

A slice of results may be selected. So far, it may appear that the results are returned as lists. This is not the case; they are actually iterable result sets. Subsets may be obtained through slicing. The chosen results are obtained using the SQL `LIMIT` and `OFFSET` directives. This allows a limited subset to be efficiently returned from a large result set, without having to transfer the query.

```
def test_query_slice():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    s2 = Student(username="doub", full_name="Doug McBride")
    s3 = Student(username="amy", full_name="Amy Woodward")
    for s in [s1, s2, s3]:
        session.save(s)
    sliced = session.query(Student)[1:3]
    assert [s2, s3] == list(sliced)
    assert [s2, s3] != sliced
```

Slicing a single element does not return a result set; it immediately returns the requested element:

```
def test_query_results_with_index():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    s2 = Student(username="doub", full_name="Doug McBride")
```

```
    session.save(s1)
    session.save(s2)
    f = session.query(Student)[0]
    assert s1 == f
```

This previous test's data is used repeatedly. Extracting the method `prepare_two_students()` leads to the following code:

```
def prepare_two_students(session):
    s1 = Student(username="jeff", full_name="Jeff Younker")
    s2 = Student(username="doub", full_name="Doug McBride")
    session.save(s1)
    session.save(s2)
    return (s1, s2)

def test_query_results_with_index():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student)[0]
    assert f == s1
```

The methods `all()`, `first()`, and `one()` are used to immediately select portions of a result set. `all()` returns all the results as a collection:

```
def test_query_results_all():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).all()
    assert f == [s1, s2]
```

The `first()` method returns the first element from a result set. It is equivalent to using `[0]`, but more expressive.

```
def test_query_results_first():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).first()
    assert f == s1
```

If only one result is returned by a query, then the `one()` method behaves like `first()`:

```
def test_query_results_one_with_one_result():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).filter_by(username="jeff").one()
    assert f == s1
```

If the query does not return precisely one result, then `one()` raises an `InvalidRequestError`.

```
from nose.util import assert_raises
from sqlalchemy.orm.exceptions import InvalidRequestError
...
def test_query_results_one_raises_error_with_multiple_results():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    assert_raises(InvalidRequestError, session.query(Student).one)
```

Filtering with SQL Queries

In the simplest cases, the `filter()` method works analogously to SQLObject's `select()` method. In these cases, it takes one argument that is a query expression written as a Python expression. These column names come from either the mapped class or the column listings in the corresponding table object's `c` attribute. The two queries in the following test are equivalent:

```
from sqlalchemy_ex import schema, Student, student_table
...
def test_simple_filter_expressions():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).filter(Student.username == "jeff")
    g = session.query(Student).\
      filter(student_table.c.username == "jeff")
    assert list(f) == list(g) == [s1]
```

SQLObject requires importing SQL expression constructors from the SQLBuilder library. Under SQLAlchemy, these sorts of operators are directly accessible from columns:

```
def test_sql_filter_expressions():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).filter(Student.username.like('%ef%'))
    assert list(f) == [s1]
```

`filter()` also accepts raw SQL where expressions. This allows you to produce hand-tuned queries when needed. These queries can contain variables that are expanded. The two string queries in the following test are equivalent:

```
def test_simple_literal_sql_filter_expressions():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).filter("username = 'jeff'")
    g = session.query(Student).\
       filter("username = :un").params(un="jeff")
    assert list(f) == list(g) == [s1]
```

In the second query, the string :un is expanded to jeff. This expansion performs the appropriate escapes. If you have to substitute variables into a query, then always use this form to stave off SQL injection attacks.

Hand-tuning goes even further with the from_statement() method. It accepts complete SQL from statements:

```
def test_from_statement():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    query = "SELECT * FROM student WHERE username like :match"
    f = session.query(Student).from_statement(query).\
      params(match='%ou%').one()
    assert f == s2
```

Keyword Queries

The filter_by() method is directly analogous to SQLObject's selectBy() method. The keywords are column names, and the values will be exactly matched. The usage has already been demonstrated, but here's a reminder:

```
def test_filter_by():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    f = session.query(Student).filter_by(username="jeff").one()
    assert f == s1
```

Chaining

The query() method produces a query object. Each filter method produces another query object as its result, so filter expressions can be chained together. The chained expressions are combined with a logical and.

```
def test_chained_filters():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    s2 = Student(username="jeffs", full_name="Jeff Smith")
    session.save(s1)
    session.save(s2)
    f = session.query(Student).\
      filter(Student.full_name.like('Jeff%')).\
      filter_by(username="jeffs").one()
    assert f == s2
```

Printing query objects shows the generated SQL:

```
>>> print session.query(Student).filter_by(username="fool")
```

```
SELECT student.id AS student_id, student.username AS➥
student_username, student.full_name AS student_full_name
FROM student
WHERE student.username = :student_username_1 ORDER BY student.id
```

One-to-Many Relationships

One-to-many relationships and the conjugate many-to-one relationships are specified from the one-to-many side. While foreign keys in SQLObject automatically yield instances of the appropriate type, in SQLAlchemy only the key value is available until the connection is established from the associated table. The relationship is declared through mappers. The schema used for this example is shown in Figure 9-6.

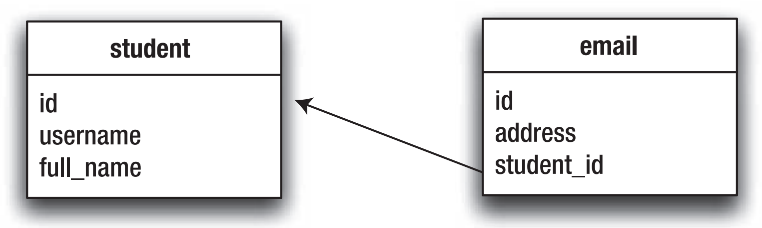


Figure 9-6. *The email table points back to the student table.*

You add the `email` table here, but the one-to-many relationship is not established yet:

```
from sqlalchemy import Column, ForeignKey, Integer, MetaData, \
  Table, String
from sqlalchemy.orm import mapper

schema = MetaData()

student_table = Table('student', schema,
                      Column('id', Integer, primary_key=True),
                      Column('username', String(16), nullable=False),
                      Column('full_name', String(64), nullable=False),
)

email_table = Table('email', schema,
                    Column('id', Integer, primary_key=True),
                    Column('address', String(255), nullable=False),
                    Column('student_id', Integer, \
                           ForeignKey('student.id'), nullable=False),
)
```

```
class Student(object):

    def __init__(self, username, full_name):
        self.username = username
        self.full_name = full_name


class Email(object):

    def __init__(self, address):
        self.address = address


mapper(Student, student_table)

mapper(Email, email_table)
```

The following test shows that the foreign key does not engender an attribute pointing back to the associated `Student` instance:

```
def test_email_doesnt_have_student_attribute():
    e1 = Email(address="jeff@not.real.com")
    assert_raises(AttributeError, getattr, e1, 'student')
```

The `mapper()` directive for the `student` table establishes the one-to-many relationship between `Student` and `Email`:

```
from sqlalchemy.orm import mapper, relation
...
mapper(Student, student_table, properties={
           'emails': relation(Email, backref="student")
})
```

The previous test now fails, and it is changed to one that verifies the existence of this attribute:

```
def test_email_now_has_student_attribute():
    e1 = Email(address="jeff@not.real.com")
    assert e1.student is None
```

The relationship can now be established from either end of the connection. On the `Student` end, the one-to-many connection appears as a list, just as with SQLObject—but there it was a read-only attribute. Here it is writable, though, and adding an object to it sets the appropriate foreign key in the added object:

```
def test_email_adding_via_one_to_many_side():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    e1 = Email(address="jeff@not.real.com")
    session.save(s1)
    s1.emails.append(e1)
```

```
    session.flush()
    assert s1.emails == [e1]
    assert e1.student == s1
```

As stated earlier, it works from the other direction, too:

```
def test_email_adding_via_many_to_one_side():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    e1 = Email(address="jeff@not.real.com")
    session.save(s1)
    e1.student = s1
    assert s1.emails == [e1]
    assert e1.student == s1
```

Elements can also be deleted from the joined attribute as if it is a normal list. Were the foreign key in the `email` table `nullable`, then the following test would work:

```
def test_email_removing_via_many_to_one_side():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    e1 = Email(address="jeff@not.real.com")
    session.save(s1)
    s1.emails.append(e1)
    session.flush()
    del s1.emails[0]
    session.flush()
    assert s1.emails == []
```

Many-to-Many Relationships

As with one-to-many relationships, creating the simplest many-to-many relationships is a little more involved than with SQLObject, as the intermediate tables must be explicitly described.

The schema described here is pictured in Figure 9-7.

```
course_table = Table('course', schema,
                     Column('id', Integer, primary_key=True),
                     Column('name', String(64), nullable=False),
)

enrolled_assc_table = \
  Table('enrolled_assc', schema,
    Column('student_id', Integer, ForeignKey('student.id')),
    Column('course_id', Integer, ForeignKey('course.id')),
)
...
```

```
class Course(object):

    def __init__(self, name):
        self.name = name
...
mapper(Course, course_table, properties={
    'enrolled': relation(Student, secondary=enrolled_assc_table)
})
```

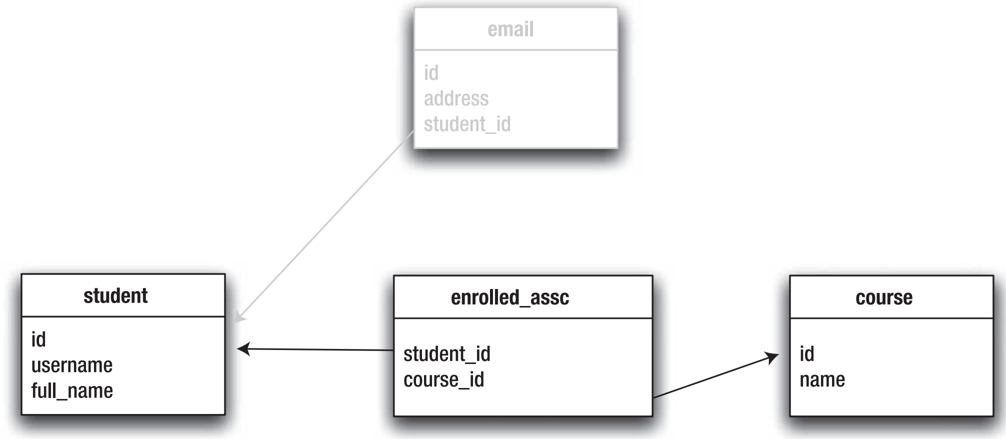


Figure 9-7. *Students are enrolled in courses.*

The keyword secondary indicates that this is a many-to-many relationship. Unlike SQLObject, it is not necessary to describe the details of this intermediate class when the relationship is declared, as this has already been done in the table definition.

The preceding declaration only creates a link from the Course object to the Student object. To make this link from the Student object back to the Course object, the backref keyword must be used:

```
mapper(Course, course_table, properties={
    'enrolled': relation(Student,
                         secondary=enrolled_assc_table,
                         backref='enrolled')
})
```

The relationship can now be viewed from either side, as shown in the next test. The next few tests require a pair of Course instances, so you'll create a method to prepare the needed test data.

```
from sqlalchemy_ex import Course, Email, schema, Student, student_table
...
def prepare_two_courses(session):
    c1 = Course('Modern Algebra')
    c2 = Course('Biochemistry')
```

```
    session.save(c1)
    session.save(c2)
    return (c1, c2)

def test_enrolled_adding():
    session = session_from_new_db()
    (s1, unused_s2) = prepare_two_students(session)
    (c1, c2) = prepare_two_courses(session)
    s1.enrolled.append(c1)
    c2.enrolled.append(s1)
    session.flush()
    assert Set(s1.enrolled) == Set([c1, c2])
    assert c1.enrolled == [s1]
    assert c2.enrolled == [s1]
```

Querying Relations

SQLAlchemy provides better support for querying relations than does SQLObject. This is done through the `join()` query method:

```
def test_select_student_by_course():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    (c1, c2) = prepare_two_courses(session)
    s1.enrolled.append(c1) # Course "Modern Algebra"
    s2.enrolled.append(c2) # Course "Biochemistry"
    session.flush()
    f = session.query(Student).join('enrolled').\
             filter(Course.name=="Biochemistry").one()
    assert f == s2
```

Deleting

Deletions are scheduled using the session's `delete()` method:

```
def test_delete_student():
    session = session_from_new_db()
    (s1, s2) = prepare_two_students(session)
    session.flush()
    session.delete(s1)
    students = session.query(Student).all()
    assert students == [s2]
```

By default, deletes don't cascade. Deleting an entity related to others leaves orphans and dangling foreign keys. Cascading deletes are a property of a relation, and are declared through the `cascade` keyword:

```
def test_delete_cascade():
    session = session_from_new_db()
    s1 = Student(username="jeff", full_name="Jeff Younker")
    e1 = Email(address="jeff@not.real.com")
    session.save(s1)
    s1.emails.append(e1)
    session.flush()
    session.delete(s1)
    session.flush()
    email = session.query(Email).all()
    assert email == []
```

When setting up delete cascades with bidirectional relations, you should pay close attention to which object is the desired parent.

Going Further with SQLAlchemy

I've tried to cover the absolute minimum that you need to know to work with SQLAlchemy. An entire book could easily be written about it.

Fortunately, it has wonderful documentation. It is concise and clear, but also voluminous. Topics worth examining further include the following:

- Building SQL expressions
- Using raw connections as a better DBI layer
- Returning custom collections from relations
- Loading strategies
- Mapping classes onto arbitrary select results
- Mapping classes onto multiple tables
- Table inheritance

Building the Database

Agile development entails short iterations. Updating databases with short iterations requires automation. There is no time to hand-tweak changes or adjust between the development environments, the QA environments, and the production environments. The only sensible way to do this is to use the same system throughout.

The database consists of the schema and the test data. Database setup must include both—however, the schema and the test data must be separable. The schema will go to QA and production, but the test data will not. The same mechanisms used to produce a database in development must be used to produce the database in QA and production. Otherwise, the changes produced in the first part of the development pipeline will have to be adapted to the mechanisms and processes used in later parts. This is a recipe for error, delay, and wasted effort.

The database schema is part of the source code. The code depends upon having a specific schema version. If application code can't access the database, then the data stored there may be worthless. Without a database running the appropriate schema version, the code is useless, and without appropriate code, the information in the database is useless; the two are inextricably intertwined. Since the database schema is part of the code base, it is owned collectively, and it is versioned.

Even though many more modifications happen to the database, using the same migration mechanisms throughout means that less operational effort must be expended. Martin Fowler has mentioned a project that had 30 developers and 1 DBA. He has also stated that many projects reduce their number of dedicated DBAs by substituting developers with database skills.

Much of the work involved with agile database development has focused on organizations in which the database is closely coupled to a single application. Some production environments have databases that serve many applications. Old financial institutions are one such case, as are potentially medical records systems. In these cases, the database schema may not travel along with a single application. There has been less work done in these areas, but the techniques developed to manage incremental database change can still be applied to them.

Testing

Programs that interact with databases have common elements, and testing each requires a different approach. These elements include

- Application–mapping layer interactions
- Mapping layer–database interactions
- Functional interactions between the application and the database
- Embedded code
- Database migrations

Application–mapping layer interactions are often addressed most easily through normal unit-testing isolation techniques; mapped objects can frequently be replaced with impostors. When testing larger subsystems, fake databases may be useful, although the existence of embedded databases and in-memory databases lessens the need for these.

Mapping layer–database interactions often benefit from using a real database of some sort. Behavioral differences between various kinds of target databases can be identified or verified before integration. Doing this requires running instances to be available to each and every developer. These days, a multi-CPU desktop has more than enough horsepower to run several virtual machines hosting "real" databases such as Oracle, Microsoft SQL Server, or Sybase. For basic sanity checking of the mapping layer, it is usually sufficient to use something like SQLite.

■ **Warning** No development work should ever be done against production databases. This is a recipe for disaster. Mistakes can easily destroy production data; successful modifications remove the database from a known state, and this has the potential to ruin automation. If a problem can't be replicated in development or QA, that suggests there is something wrong with the development and QA resources.

Functional interactions between the application and the database are addressed by using real databases. As noted previously, VMs are useful for this. Indeed, I know of several organizations in which a scaled-down version of the entire production network runs on each developer's desktop. Over the next few years, I expect this sort of full environment simulation to become more common.

Embedded code refers to code that runs in the database. Triggers and stored procedures are the most frequently used kinds of embedded code. The only way to test this code is with a live database.

Database migrations must be tested against their target databases. Those migrations must be tested against both the schemas and realistic sets of data. How this is done constitutes much of the remainder of this chapter.

Refactorings

Refactorings are modifications to the database that improve its structure. As with code refactorings, these are incremental changes. They can be applied to the existing database to achieve a desired final state. Unlike source refactorings, they have to take data into account, too.

If you're lucky, then your database can be brought down completely while your software is upgraded and the current database is refactored. If you work in a 24/7 environment or you have many applications working with a single database, then this is often not a possibility. In these cases, refactorings have to be performed in two steps.

The first step adjusts the database so that both the old and new versions of the application continue to work with the new database. This is referred to as the transition period. During this time, the applications depending on the old schema are upgraded.

Once the applications have been upgraded, compatibility with the old schema is no longer required. It is safe to transform the database to the completed state.

Consider renaming a column from `Foo` to `Bar`. When the change is applied, a new column, `Bar`, is added. Either the application code or database machinery mirrors changes between `Bar` and `Foo` so that old code continues to operate. Once all running applications reference `Bar`, the mirroring mechanism is turned off, and the column `Foo` is removed.

Almost all live production database environments are upgraded incrementally, so refactorings fit into the DBA's natural operational model.

"But I run a Java cluster on server Foo, and it switches deployed versions all at once!" No it doesn't. Every Java clustering mechanism I have seen switches over to the new version after the old connections terminate. Connections to each machine will terminate at different rates. This means that every member of the cluster switches to the new version of your application at different times. The result is that, often for a few seconds, some proportion of your cluster is running at version n, and some proportion of your cluster is running at version n+1.

Migrations

All developer databases should be synchronized. This can be done in one of two ways: either each developer's instance is updated from a central location, or the database is reconstructed by the build. These two are not mutually exclusive, either.

I prefer the second option, in which the database is reconstructed by the build. The migration of the central source of truth needs to be done, and so it will need to be in a replicable

manner. If it can be done once, then it can be done many times, and only one mechanism will be required.

Over the last few years, a consensus has started to emerge about how database migrations should be performed. The database records what schema version it is running. Developers write a set of instructions describing how to get from one version to the next. A tool consults the database version and the desired version, it determines which set of instructions must be applied, and then it applies them.

These instructions are never applied by hand—only by automation. They are applied to production en masse at release time. Generally, reverse scripts are supplied so that the production database may be rolled back if needed.

The Instructions

There are two broad systems for generating the migration instructions:

> With *explicit migrations*, the developer declares which steps will be taken. In some cases, this is done with XML declarations. In others, it is done with a DSL (Rails Migrations is an example). Sometimes it is done in the application language code, and often it is done in raw SQL. The first three techniques have the advantage of being database independent. The last is not, but it has the advantage of giving direct access to many database features, including applying security.

> With *derived migrations*, the instructions are derived from the difference between the desired schema and the current schema. Sometimes the desired state is encoded in the full schema. More often, it is in a DSL, with XML being one of the more common formats. In some systems, the difference is derived by comparing the desired schema with the database itself.

Derived migrations are appealing, but in practice experienced developers seem to be very wary of them. Even if the derived migrations work, data migration code must still often be written. A framework for running this code must be supplied, and this is the same framework that is necessary for running explicit migrations. If you have one, why complicate it with the other? Explicit migrations also give the opportunity for migration procedure code reviews.

Numbering Migrations and Playing Them Back

In the simplest case, the migration scripts are numbered in a monotonically increasing integer sequence (1, 2, 3, etc.). The database records the most recently applied script. All scripts below that are assumed to have been applied, and those above have not. When a migration happens, all the scripts between the current version and the desired version are played back, and the last one is recorded.

The simple numbered sequence works for small groups working on a single codeline. When those two assumptions break down, so does the sequence numbering. The numbers are now a scarce resource, and developers must arbitrate access to them. When merges happen, duplicates collide and the migrations have to be renumbered, and heaven help you if they have been applied to shared resources. The potential for error quickly becomes large.

The solution is to use numbers with a low chance of collision: timestamps. They are monotonically increasing integers, and there is a small chance that two people will choose exactly the same second to create a migration file. The migration system also needs to know

nothing about what they represent, as they're just a fancy kind of integer. A typical timestamp might be `20080204080953` (Feb, 4, 2008, 08:09:53.)

That doesn't completely solve the merge problem, though. Consider the case in which one branch has already been applied to a database, and another branch is merged in. Both were under development at the same time, so they have migrations with intermingled timestamps. The odds are that these migrations are independent, so you should be able to play them back successfully.

Problems happen when some of the newly merged migrations are below the current version. These will not be played back when a new migration is attempted. This can be solved by tracking all of the applied revisions instead of just the most recently applied revision. When a migration happens, the desired version is determined, the applied migration list is consulted, and then all unapplied migrations below the desired version are applied to the database.

Where to Put the Migration Mechanism

Applying migrations can be viewed as part of the installer, part of the application, or the duty of a special application that just manages database upgrades. It all depends on the application that you're using and its intended purpose. I don't have firm feelings except in the case of clustered applications. In these environments, coordinating deployment is a major headache, and I feel that migration duties belong with the application or with a special-purpose database migration management application. No matter where you put it, a mechanism is required to manage migration attempts.

DBMigrate: A Migration Mechanism

Writing the migration mechanism itself is painful. While not seeming terribly complicated, it has lots of edge cases that need to be addressed. This problem has been tackled in the Ruby world with Rails Migrations. In the Java world, PatchDB is a notable example, and there are many others. The Python world has had no such mechanism . . . until now. In the course of working for my employer I had to write one, as did an acquaintance of mine. I've taken aspects of my code and his, and I've published DBMigrate.

Using DBMigrate

DBMigrate is installed via `easy_install dbmigrate`. Migrations and test data are written as Python packages. These migrations can be applied through the following:

- Command-line tools
- Setuptools directives
- Embedding the engine within your program
- Unit tests

The tool supports test data importation. Applied migrations are tracked individually rather than using a single counter. The application supports bootstrapping and complete tear-down of a database. Migrations are explicit, and they are written as raw SQL, Python functions, or a mixture of both. Different kinds of databases can have different migrations;

MySQL may have one set of migrations while SQLite has another. In this case, MySQL might set user permissions. SQLite does not have user permissions, so these are skipped.

Starting from Scratch

The database must be created before it can be used. With SQLite, this isn't a problem—the file is the database. However, with other databases, DBMigrate must be able to connect as a user that has permission to create databases, and in many cases, to grant privileges. DBMigrate calls this user the *admin user*; the admin user has an associated admin password. This is distinct from the application user that will attach to the database in production or during testing.

Throughout the application, the following keywords are referenced:

`scheme`: The database connection scheme (e.g., `sqlite` or `mysql`).

`admin_user`: The database user with rights to create a database, create users, and grant rights.

`admin_pw`: The admin user's password.

`user`: The database user that the application connects as. This user may not exist until the migrations are run. If this is omitted, then it is assumed to be the same as `admin_user`.

`pw`: The application user's password. If this is omitted, then it is assumed to be the same as `admin_pw`.

`db`: The name of the database to be created.

`host`: The name of the host on which the database server runs.

`port`: The port number on which the database server listens.

`socket`: The path to the socket that the database listens on.

`versiontable`: The name of the table containing the applied revisions.

These values are passed to the migration scripts in a dictionary. This is the set of *expansions* for a database.

Creating Migrations

The first migration must always set up the records table. Migrations are stored in a table.

```
$ python ./setup.py make_migration --package apptest.db.schema➥
--name create_db
```

```
Migration apptest.db.schema.migrate_20080218151301_create_db created.
```

```
$ more apptest/db/schema/migrate_20080218151301_create_db.py
```

```
# Migration template created by DBMigrate at 2008/02/18 at 15:13:01 UTC.

migration = []
```

Migrations are specified as a list of atoms. The atoms are tuples. The first component of the atom is a single SQL statement that performs an upgrade. The second component is a single SQL statement that undoes the first operation. Either one may be an empty string or None.

The atoms are applied in order from first to last when upgrading. They are applied in reverse order when downgrading.

A sample migration to create the student table from earlier in this chapter follows. This is one of those cases where I feel that breaking convention for readability is worth it.

```
migration = [("""
        CREATE TABLE student (
           id INTEGER PRIMARY KEY AUTO_INCREMENT,
           full_name VARCHAR(64) NOT NULL,
           username VARCHAR(16) NOT NULL
        )
    """, """
        DROP TABLE student
    """),
]
```

Migration strings are expanded before they are executed. This is done with Python string expansion using named parameters such as %(*foo*)s. The precise set of expansions depends upon the kind of database being constructed. SQLite uses only the minimum set of expansions: scheme and versiontable. Databases with multiple accounts will always expand user and pw. Database servers with multiple databases also expand db.

```
migration = [("""
        CREATE TABLE %(db)s.student (
           id INTEGER PRIMARY KEY AUTO_INCREMENT,
           full_name VARCHAR(64) NOT NULL,
           username VARCHAR(16) NOT NULL
        )
    """, """
        DROP TABLE %(db)s.student
    """),
]
```

Migrations can also be functions. These functions receive a SQLAlchemy connection argument. The function optionally accepts an expansions dictionary. Migration functions and migration strings can be freely intermixed.

```
def my_data_migration_up(connection, expansions):
    # This just happens to accept an expansions dictionary
    pass

def my_data_migration_down(connection):
   # This doesn't
   pass

migration = [(my_data_migration_up, my_data_migration_down)]
```

Different migrations can be specified for different database schemes. These schemes correspond to the schemes used in SQLAlchemy database URIs. In this case, `migration` is a dictionary instead of a list. The keys correspond to the database's URI scheme. For example, if the URI for the database was `mysql://localhost/db`, then the scheme would be `myqsl`. The default migration is used when there is no matching migration. It is keyed with _.

```
generic_migration = [("""
        CREATE DATABASE %(db)s
    """, """
        DROP DATABASE %(db)s
    """),
   ("""
        CREATE TABLE %(db)s.%(versiontable)s (
           id INTEGER PRIMARY KEY AUTO_INCREMENT,
           package VARCHAR(64) NOT NULL,
           revision INTEGER UNSIGNED NOT NULL
        )
    """, """
        DROP TABLE %(db)s.%(versiontable)s
    """),
]

sqlite_migration = [("""
        CREATE TABLE %(versiontable)s (
           id INTEGER PRIMARY KEY AUTOINCREMENT,
           package VARCHAR(64) NOT NULL,
           revision INTEGER UNSIGNED NOT NULL
        )
    """, """
        DROP TABLE %(versiontable)s
    """),
],

migration = {'_': generic_migration, 'sqlite': sqlite_migration}
```

The first migration always creates the database and migration schema. Templates for these migrations are available in `db.migrate.templates`. Currently, there are templates for SQLite, MySQL, and PostgreSQL, and one suitable for use under Pylons.

Manually Migrating a Database

Databases are manually migrated using `setup.py`. Different parameters are supplied depending on the database being migrated. The following command shows a MySQL database being upgraded to the most recent version:

```
$ python ./setup.py dbmigrate --scheme mysql --admin_user root➥
--admin_pw ROOT_USER_PW --user dbu --pw dbpw➥
--host localhost -v --db mydb my.app.schema
```

```
Upgrade from revision 0 to revision 20080219120356
Applying my.app.schema.migrate_20080218151301_create_db
Applying my.app.schema.migrate_20080218192701_create_student
Applying my.app.schema.migrate_20080219120356_create_email
```

The database can be torn down using the `--revision` flag:

```
$ python ./setup.py dbmigrate --scheme mysql --admin_user root➥
--admin_pw ROOT_USER_PW --user dbu --pw DB_PW➥
--host localhost -v --db mydb --revision 0 my.app.schema
```

```
Downgrade from 20080219120356 to revision 0
Applying my.app.schema.migrate_20080219120356_create_email
Applying my.app.schema.migrate_20080218192701_create_student
Applying my.app.schema.migrate_20080218151301_create_db
```

As with any Setuptools command, commonly used options can be stored in the `setup.cfg` file. The section heading is `[dbmigrate]`. For the preceding command, the corresponding `setup.cfg` is the following:

```
[dbmigrate]
    scheme=mysql
    admin_user=root
    admin_pw=ROOT_USER_PW
    db=mydb
    user=dbu
    pw=DB_PW
    host=localhost
```

Minus the verbose flag, `-v`, the previous installation commands are now the following:

```
$ python ./setup.py dbmigrate my.app.schema
$ python ./setup.py dbmigrate --revision 0 my.app.schema
```

When more than two packages are passed to DBMigrate, the migrations are interleaved based on their timestamps. The ordering between identical timestamps is undefined; they may be applied in any order.

```
$ python ./setup.py dbmigrate -v my.app.schema my.test.schema
```

```
Upgrade from revision 0 to revision 20080219120534
Applying my.app.schema.migrate_20080218151301_create_db
Applying my.app.schema.migrate_20080218192701_create_student
Applying my.app.testdata.migrate_20080218192734_populate_student
Applying my.app.schema.migrate_20080219120356_create_email
Applying my.app.testdata.migrate_20080219120534_populate_email
```

Running DBMigrate with Unit Tests

Running unit tests requires a minimal `setup.cfg`. `scheme` is the only required parameter for all databases; in general, only the minimum amount of information required to create an administrative connection is needed. For SQLite, nothing is required; for MySQL, only `scheme=mysql`, `admin_user`, and `admin_pw` are required.

A `setup.cfg` for MySQL might read as follows:

```
[dbmigrate]
    scheme=mysql
    admin_user=root
    admin_pw=ROOT_USER_PW
    host=localhost
```

The values `user`, `pw`, and `db` are ignored by the unit tests, as DBMigrate creates random values for them. This results in a unique database; these randomly chosen values are still passed to the migration scripts in the expansions dictionary.

Within a unit test, migrations are applied using `dbmigrate.DBTestCase` in the case of unittest. The method `connect_application()` is called from `setUp()`, and the method `disconnect_application()` is called from `tearDown()`. If these methods are not supplied, then no error results.

```
from dbmigrate import DBTestCase

class MyTestCase(DBTestCase):
    """Database test case using migrations framework"""

     # One or more sets of migrations
    migrations = ['my.app.schema', 'my.app.testdata']

    def connect_application(self, uri, expansions):
        # The function should connect your application code
        MyApp.connect(uri)

    def disconnect_application(self, uri, expansions):
        # This function should disconnect your application code
        MyApp.disconnect()

    def test_method(self):
        # this would be a test method if this were real code.
        MyApp.run()
```

Running DBMigrate from Your Program

Your program may need to control the setup or tear-down of a database at install time or runtime. In these cases, the migration framework can be embedded and run from your application code. The migration engine is run with a dictionary of connection parameters and a list of packages containing schema files.

```
from dbmigrate import MigrationEngine
...
def install_database():
    config = read_your_app_config()
    params = {'scheme': 'mysql',
              'admin_user': config['db.admin_user'],
              'admin_pw': config['db.admin_pw'],
              'user': config['db.user'],
              'pw': config['db.pw'],
              'host': config['db.host'],
              'port': config['db.port'],
              'db': config['db.name'],
    }
    MigrationEngine().run(params, ['my.app.schema'])
```

The preceding code shows how the connection parameter dictionary `params` might be constructed from an application's configuration, and then how the migration engine is run.

Specific revisions may also be requested using the `revisions` keyword:

```
MigrationEngine().run(params, ['my.app.schema'],
                      revision=20080215135324)
```

The database is upgraded or downgraded to achieve the desired revision.

Summary

Database technology has become steadily cheaper and more widely available over the last ten years. Only recently has database development started to adapt to these changing realities. The result is an agile approach to database development sometimes called evolutionary database development. It is built around the assumption that databases, like software, will change.

Agile development's focus on short iterations forces this change. Traditional DBA organizations are unable to meet the rapid turnaround without large increases in staff. The only way to shorten cycles is through automation, and automation by nature pervades the entire development cycle.

As a result, database development is viewed as part of the overall software development process. DBAs and developers work closely together to understand each other's concerns as early in the development cycle as possible. The developers themselves often compose the migration scripts for the database.

ORMs are valuable tools for isolating application code from the details of the underlying database. They map between application objects and database structures such as tables, columns, and rows. Because database access is mediated by normal objects, testing can be performed using normal techniques. The two most common Python ORMs are SQLObject and SQLAlchemy.

In an agile environment, database upgrades must be performed through automation. This automation must be consistent and testable, and is often done through schema migration systems. These store the schema version in the database and compare it to the version required by the code. If the two disagree, then a series of migration scripts are applied to bring the database into conformance.

Agile database development is very much on the cutting edge of agile development. There is room for many new tools and new practices. It is problematic because the development involves external applications and additional special-purpose languages working in conjunction with the application code.

Another cutting edge area is web development, which has similar issues. Today's rich web applications are built around JavaScript running in the user's browser. Compatibility must be maintained against a wide variety of client platforms. Somehow this code and its interactions with the Python application must be verified—which is the subject of the next chapter.

CHAPTER 10

Web Testing

The World Wide Web was the killer app for the Internet. In the course of less than a decade, it went from a simple document-sharing system for physicists to ubiquity. In 1994, if you'd said to someone that six years later billboards hawking milk would have URLs plastered on them, you would have been asked if you'd seen your psychiatrist recently and if she'd considered upping your dosage. Nevertheless, six years later there *were* URLs on billboards hawking all manner of consumer wares.

To say that the Web grew quickly is an understatement. It grew quickly, and it grew from the ground up. The technologies composing it were not planned out. They arose from need and circumstance. To put it more frankly, the Web as we know it today is a hodgepodge of different technologies that have been hacked together with the digital equivalent of bubblegum, spit, and baling wire. Afterward, standards bodies come through and codify the things that held together, but by that time everyone else has rushed on to the next set of problems.

Despite this madcap development, the Web has a very simple basis. Every application must use the same technologies to talk to the browser, so web applications have gross similarities in structure. These similarities give rise to repeated solutions to problems, which in turn means repeated testing methods. This is true of both unit testing and functional testing, but in this chapter I'll be demonstrating unit testing tools.

Really Simple Primer

At its simplest, the Web is a document format combined with a notation identifying these documents, and a protocol for using those identifiers to retrieve the documents.

- The document format is *HTML (Hypertext Markup Language)*.
- The document identifiers are called *URIs (Universal Resource Identifiers)*. When used to locate documents on a network, they are called *URLs (Universal Resource Locators)*.
- The network protocol is *HTTP (Hypertext Transfer Protocol)*.

All three are based on ASCII text rather than binary encodings. This makes them easy to manipulate with text-based tools such as text editors or telnet clients.

Web browsers are programs that retrieve documents via URLs, render the HTML, and then allow users to follow the URLs included in the retrieved documents. At its simplest, a browser follows a well-defined series of steps:

1. The user supplies a URL to the browser.
2. The browser uses the URL to locate the server containing the required document.
3. The browser requests the document from the server by HTTP.
4. The server returns the document to the browser.
5. The browser renders the document and presents it to the user.

This system allows for a limited amount of interaction from the user. The document may specify data to be retrieved from the user and a method for sending the results back to a server. This data is still sent back via HTTP. The result is yet another document.

HTML was originally intended to describe the content of a document, and not its formatting, but it was quickly forced into that role. *Cascading Style Sheets (CSS)* was created to restore this separation.

HTML is based on a document format called SGML (Standard Generalized Markup Language). SGML eventually begat a simpler markup language called XML (eXtensible Markup Language), which has become wildly successful for representing many kinds of data.

HTML

HTML is a text format. Ideally it describes the contents of a document, not how that document is to be rendered. In reality, this ideal is rarely met, and the elements of a form are often used for layout. Crucially, HTML documents also describe how they connect to other documents.

This is a simple HTML document:

```
<html>
   <head>
       <title>My Favorite Comics</title>
   <head>
   <body>
        I love XKCD and PVP.
    </body>
</html>
```

An HTML document can be viewed as a tree. The opening tag `<foo>` defines a node named “foo.” It is terminated by the closing tag `</foo>`. The nodes are referred to as elements. Elements nest, but they do not interleave. If a tag contains no other elements, then the opening and closing can be combined, as in `<foo/>`.

Elements can also contain key/value pairs called *attributes*. Here the input tag has the type `text` and the name `comicname`: `< input type="text" name="comicname" />`.

SGML, from which HTML was derived, is a vast standard developed by committee. It was far from simple, and its parsers had to be very complete and strict in their interpretations.

HTML is a limited derivative of SGML with a very narrow problem domain: displaying simple documents on a network. HTML parsers were intended to be very forgiving so that slightly inaccurate documents created by relatively naive users could be successfully

presented. While this do-what-I-mean approach is in the spirit of Postel's Law,[1] it has introduced much ambiguity, and has resulted in a situation where no two web browsers render things in exactly the same way.

CSS

CSS describes how HTML documents are to be rendered. It is the result of an effort to remove formatting information from HTML documents. CSS binds HTML tags to formatting directives.

XML

The wild success of HTML and the relative failure of SGML gave birth to an effort to simplify SGML. This led to XML. The preceding description of HTML tells you most of what you need to know about XML syntax.

In the late '90s, XML was hyped beyond all belief. Vendors were suggesting that it would solve all data interchange problems, when clearly this was not the case. Despite this failure to deliver on the hype, I feel that it has been underappreciated for what it really does.

It provides a common syntax for structuring data, essentially doing for file formats what ASCII did for character sets. Having a common character representation vastly increased the portability of programs across computer systems, but it didn't solve all data interchange problems. It just allowed the focus to be raised to a new level of abstraction. XML does the same for data by supplying a universal syntax.

URI and URL

The URI format identifies documents unambiguously. Once obscure, it can now be seen even on billboards for toilet paper. A URI has four parts, organized as follows:

scheme **:** *hierarchical part [* **?** *query] [* **#** *fragment]*

The scheme identifies the kind of resource, and it determines how the other three parts are interpreted. Common schemes include the following:

- `http`, for web pages
- `https`, for encrypted web pages
- `file`, for files on the local system
- `mailto`, for e-mail addresses

The hierarchical part is separated from the scheme by a colon, and it is mandatory. A question mark separates the hierarchical portion from an optional query. It contains nonhierarchically organized information. The fragment is separated from these parts by a pound sign, and it serves as a secondary index into the identified resource. The following URI shows all the parts:

```
http://www.theblobshop.com/theguide?chapter=6times9#answer
```

---

1. Postel's Law is "Be conservative in what you do; be liberal in what you accept from others." See, for example, `http://ironick.typepad.com/ironick/2005/05/my_history_of_t.html`.

When a URI contains the information necessary to locate a resource, it is referred to as a URL. The two terms are often used interchangeably.

HTTP

HTTP is the network protocol that the Web is built on. It is defined in RFC 2616. In this protocol, a client initiates a connection to a server, sends a request, receives a response, and then disconnects. The server is not expected to maintain state between invocations.

A request consists of the following:

- A command
- The URI the command operates on
- A message describing the command

A response consists of the following:

- A numeric status code
- A message describing the response

The request and response messages share the same format. It is precisely the same format as that used to represent letters in e-mail, and it is defined in RFC 822. The message consists of a set of headers followed by a blank line and then an arbitrary number of data sections. There is one header per line, and each header is just a name/value pair. The name is on the left, the value is on the right, and they are separated by a colon.

JavaScript

JavaScript is not Java. It is not Java-light. It has nothing to do with Java. It is a dialect of a standardized language called ECMAScript, which is defined in the document ECMA-262.

JavaScript is a dynamically typed, object-oriented, prototype-based language. It has a C-based syntax, but its object model is much closer to that of Python, and Python programmers will find themselves at home.

JavaScript executes within the browser, and each browser has its own slightly different implementation. JavaScript programs manipulate a tree-shaped data structure representing the HTML document they reside in. Changes to this document are reflected on the screen. JavaScript programs can also send data back and forth to the server from which they were retrieved.

A display model that can be easily manipulated, combined with two-way network communications, has given rise to a programming paradigm called *Ajax (Asynchronous JavaScript and XML)*. You can use Ajax techniques to create web pages that behave much like local applications.

Web Servers and Web Applications

Web applications run on both the client that displays the pages and the server that delivers them, yet almost all applications start with the server. There is wide variation in how the applications are implemented.

At one end are simple scripts executed by the web server. The web server and scripts typically communicate using the Common Gateway Interface (CGI) defined by RFC 3875. At its heart, this standard defines a few more request headers describing the HTTP conversation. These are passed to your script, and the server expects your script to send back a few more headers. The new HTTP request is passed into your script via `stdin`, and the server reads the response message from your script's `stdout`.

The odds are that you will never deal with CGI at such a low level; all languages that I can think of provide libraries for handling these nuts and bolts. In Python, this library is named `cgi`.

At the other extreme are full stack applications. These implement everything from the web server to the application logic. They are often seen in shrink-wrapped applications, or with applications that act as platforms for other applications. One example in Python is the Plone content management system.

Between the two extremes are applications written with web application frameworks. These typically run on top of different web servers. These frameworks support writing complex applications, providing solutions for common problems. Typical features are

- Form validation and data conversion
- Session management
- Persistent data storage
- HTML templating

Common Python application servers include

- Zope
- Django
- Google App Engine
- Pylons
- Turbogears

These days, most applications of any appreciable size are written with web application frameworks. These frameworks run on top of some kind of a web server, such as Apache, IIS, or the Python-based Twisted.

Application frameworks typically have large startup costs connected to the extensive services they provide, so running them from CGI isn't feasible. The delay between the user's request and the application's response would be too long. Instead they connect to web servers through different mechanisms.

These mechanisms fall into two broad categories. In one, the application runs as part of the web server itself, and in the other, the application runs in a separate process and the web server forwards requests and responses to this process.

When an application framework runs as part of a web server's process, there is often little configuration to be done. The application often has direct access to the web server's internal state and its optimized services. The problem is that you're engaging directly with the web server's environment. This can lead to strange interactions, particularly when other applications are also running in the server's address space.

There are as many ways of doing this as there are web servers, since each different kind has its own extension interfaces. With Apache, this functionality is provided by the Apache plug-in `mod_python`.

THE PROBLEM WITH OCCUPYING ANOTHER'S SPACE

I once spent days trying to determine why a Python application was failing when running under `mod_python`, but succeeding from its test environment. It used the SQLObject object-relational mapping layer (see Chapter 9) in combination with the MySQLdb back end. The application would access the database layer, and then simply die without sending a response. There were no messages in the logs, there were no stack traces, and there were no core dumps.

Tracing the calls at the system level led to the discovery that PHP was loading a custom version of the dynamically linked MySQL client libraries. When MySQLdb attempted to load the client libraries, it was instead linked with the PHP version. The PHP version was incompatible at a very low level, and the calls to the database died silently.

Luckily, PHP was not required for the operation of the production system, and I was able to turn off the `mod_php` plug-in with impunity.

The alternative approach is running the application framework in another process. The web server passes requests and responses to and from the external process. Once again, there are multiple ways of accomplishing this, but in this case there is also a standard mechanism called FastCGI.

To make things worse, every application framework used to have its own method for interfacing to each web server. Even if two different frameworks both had FastCGI adapters, each was configured in a different way. Having *m* web server interfaces and *n* web servers leads to $m \times n$ combinations; or to put it more succinctly, it resulted in a big mess.

What happens when you want to connect multiple web applications to a single web server? What if you want to set up more than one application running under the same application framework? These used to be significant problems, but they've been solved within the last few years.

WSGI

The Web Server Gateway Interface (WSGI; pronounced *whiskey*) defines a simple interface between web servers and Python web applications. It is defined in PEP 333. Adapters are written from the web servers to WSGI, so applications only have to support a connection to WSGI. Over the last few years, WSGI has become ubiquitous. On the server side, it is supported by Apache, CherryPy, LightHTTPd, and Zope, among others. On the app server side, it is supported by CherryPy, Django, Pylons, Turbogears, TwistedWeb, and Webware, to name a few.

The interface is similar in concept to Java's Servlet interface. While servlets are designed for implementing any kind of network protocol, WSGI is focused on HTTP.

There are two parties in each WSGI conversation: the gateway and the application, with the gateway representing the web server. The application is a callable, and I'll refer to it as `application`. The interaction can be summarized as follows:

1. The gateway calls `application` passing an `environ` dictionary and a `start_response` callback. The dictionary `environ` contains the application's environment variables.

2. The application processes the request.

3. The application calls `start_response`, passing the response status and a set of response headers back to the gateway.

4. The application returns the response contents as an iterable object.

In the first step, the gateway calls `application(environ, start_response)`. The `application` object must be a callable, but it may be a function, a class, or an instance. The method the gateway calls for each of these is shown in Table 10-1.

Table 10-1. *Call Equivalents*

application Is a(n) . . .	application(environment, start_headers) Is Equivalent to . . .
Function or method	`application(environ, start_headers)`
Class	`application.__init__(self, environ, start_headers)`
Object	`application.__call__(self, environ, start_headers)`

In the third step, the `application` object calls `start_response(status, headers)` when it is ready to return HTTP results. This must be done before the last result is read from the iterator returned by `application(environ, start_headers)`.

In the fourth step, the returned sequence may be a collection, a generator, or even `self`, as long as the returned object implements the `__iter__` method.

Using the write Callback

Some underlying web servers read the application's results in a different way. They hand the `application` object an output stream, and instead of returning the results, the `application` object writes the results to this output stream. This stream is accessed through the `write(data)` callback, which is returned from `start_response(environment, headers)`. In this case, the calling sequence is as follows:

1. The gateway calls `application(environment, start_response)`.

2. The `application` object calls `write = start_response(status, headers)`.

3. The `application` object writes the results: `for x in results; write(x)`.

4. The `application` object returns empty results: `return [""]`.

WSGI Middleware

In this chapter, I will use the term *middleware* in the limited sense defined by WSGI. These components are both WSGI gateways and WSGI applications. They are shimmed between the web server and the application. They add functionality to the web server or application without needing to alter either. They perform duties such as the following:

- URL routing
- Session management
- Data encryption
- Logging traffic
- Injecting requests

The last two give an inkling of why WSGI middleware is important to testing. Middleware components provide a way of implementing testing spies and call recorders. These can be used to create functional tests. The underlying web server can also be completely replaced by a test harness that acts as a WSGI gateway. This bypasses the need to start a web server for many kinds of tests.

Testing Web Applications

Web testing breaks down into the two broad categories of unit testing and integration testing. Integration testing involves multiple components being tested in concert. It requires a more complicated testing infrastructure, it distances your tests from the origin of your errors, and it tends to take more time. It is an invaluable approach with web applications, since there are aspects of many programs that can't be performed in isolation, yet because of its shortcomings, it should be used judiciously.

This returns us to the idea of designing for testability. By restructuring your program, you can limit the number of places where you have to run integration tests, and this restructuring happens to result in more maintainable programs. There is a well-defined architecture called *model-view-controller (MVC)* that facilitates this.

MVC separates the input (controller) and output (view) from the computation and storage (model). Web programs receive sets of key/value pairs at distinct intervals as input. The computation is no different than with any other software. Both of these are easily tested with techniques you've already seen in previous chapters. The real differences reside in the view.

The views generate four distinct kinds of output:

- Graphics
- Marshalled/serialized objects in text form
- Markup
- Executable content

Each has a distinct set of testing strategies.

Graphics and Images

There are multiple levels of image testing. There are two basic strategies: one is to watch the image generation process, and the other is to examine the resulting image.

The first is accomplished with testing techniques that we've already examined. The drawing library is replaced with a fake or a mock, and the resulting instructions are verified. Common sequences of primitive drawing operations are combined into larger operations. These can be verified and then used as the blocks for instrumenting larger higher-level drawing operations.

The other approach employs additional techniques. At the simplest, you can check whether something was returned, and the basic characteristics are checked without regard for the contents at all. Image generation should produce results, and it should do so without raising an error. Verifying this may be enough for some problem domains.

The image can be validated through parsing. It is passed to the appropriate image library and rendered to an internal representation. The rendering process will fail if the image is not valid. Once rendered, your graphics library may supply enough data to verify certain image characteristics. These could include the image width and height, the image size, the number of bits in the color palette, or the range of colors.

In other cases, the contents of the images may need to be verified. The simplest cases are when a known image is generated. The resulting image may be compared byte for byte against a reference image. For other kinds of images, it may be sufficient to compare certain image properties such as the center of mass, average brightness, color spectrum, or autocorrelation results. These sorts of properties are generated using image-processing libraries. Each library has unique properties and should be chosen with regard to which properties must be measured.

Vector image formats often produce instructions that may already be text or that can be easily converted to text, and they may be treated as if it they were any other kind of text document.

It may also be possible to instrument the rendering library itself. The test subject is passed to the rendering library, and the calls that it produces are verified either through logs generated by test spies or by fakes and mocks.

Markup

The output from web applications isn't strictly limited to markup documents, but they form the vast bulk of the output you'll be testing. These can be analyzed through lexical, syntactic, and semantic tools. For the simplest cases, where you just want to verify that a word was included in otherwise tested results, lexical analysis may be sufficient. In these cases, the HTML output is just text, and the entire toolbox of Python string operators may be brought to bear. Regular expressions and `string.find` are very useful in these cases.

One of the primary drawbacks of lexical testing is that it doesn't verify that the document is well formed. However, this is easily done through syntactic testing techniques. In particular, the Python standard library includes `HTMLParser` for these simple cases.

At the syntactic level, it may be enough to verify that the output is valid HTML. This can be accomplished by passing the document through the standard library's `HTMLParser`. It allows you to quickly verify that a sequence of tags is included in a page, but it tells you little about the meaning of those tags—it's a very low-level tool.

More complete parsers produce a tree representing the parsed document. The structure and relationship between nodes is available for your tests' perusal. The elements are the nodes, and they are named. Attributes are attached to the element, as are the attribute values. Child and sibling nodes can be iterated for every element. This functionality is available through the standard library's `ElementTree` package.[2] Parsing a document with `ElementTree` is easy:

```
import xml.etree.ElementTree as et
...
doc = """
<html>
   <head>
      <title>Comic Feeds</title>
   </head>
   <body bgcolor="#ffffff">
      You are not subscribed to any feeds
   </body>
</html>
"""

parsed = et.XML(doc)
```

The parsed object is an `ElementTree` describing the document. Each node contains methods for navigating the subtrees.

```
def setup(self):
    self.root = et.XML(doc)
def test_get_tag_name(self):
    root = et.XML(doc)
    assert self.root.tag == 'html'

def test_get_children(self):
    children = self.root.getchildren()
    assert children[0].tag == 'head'
    assert children[1].tag == 'body'

def test_get_attributes_from_body_tag(self):
    body = self.root.getchildren()[1]
    assert body.item() == [('bgcolor', '#ffffff')]
```

The line between syntactic analysis and semantic analysis of HTML documents is fuzzy. When writing tests, you want to know the answer to questions such as the following:

---

2. `ElementTree` was added to the standard library in Python 2.5, so it is not present in earlier versions. It still exists as an external package, and you can install it with `easy_install`. It installs into a different namespace: `elementtree.ElementTree`. It is under active development, and there have been significant improvements since it was added to the standard libraries, so it may be worth installing it even if you are using Python 2.5. In this case, it happily coexists with the standard installation.

- Are the two links to my favorite comics included in this document?
- Is the table of contents included?
- Are there three links to xkcd?

These all involve searching for specific nodes within the parsed document. The overall test pattern is the same—the document is parsed, and then the element tree is searched for the relevant tags. `ElementTree` searches are done with the `find()`, `findtext()`, and `findall()` methods. The following code finds the `title` tag in the previous example:

```
def test_find_title_tag(self):
    title = self.root.find('.//title')
    assert title.tag == 'title'
```

The `findtext()` method returns the text contained in the found tag:

```
def test_find_title_text(self):
    title_text = self.root.findtext('.//title')
    assert title_text == 'Comic Feeds'
```

If the search expression matches more than one element, then these two methods will only return results for the first one. To return all matches, you must use the `findall()` method:

```
def test_find_all_top_of_roots_children(self):
    root_children = self.root.findall('*')
    assert len(root_children) == 2
    child_tags = [x.tag for x in root_children]
    assert child_tags == ['head', 'body']
```

You may be wondering about the strange query strings that the find operations use. These are XPath queries. *XPath* is a standard format for locating nodes within an XML document. XPath is somewhat like a directory path specification. It is a very rich query language, but the `ElementTree` implements only a small subset of the full specification. Despite its limitations, it's quite usable for many testing purposes.

A summary of query components can be found in Table 10-2. The full XPath specifications can be found on the World Wide Web Consortium (W3C) web site at `www.w3.org/TR/`. Although the current version is 2.0, most XPath packages still support only 1.0 or some variant thereof. More complete XPath implementations can be found in other Python packages such as PyXML.

Table 10-2. *The XPath Operations Supported by ElementTree*

Operator	Action
`foo`	Matches an element with the tag `foo`
`*`	Matches any child tag name
`.`	Specifies the current tag; mostly used at the top level
`/`	Separates levels within the tree
`//`	Finds the next pattern anywhere in the subtree

The other solution is the package named `BeautifulSoup`. It is downloaded via `easy_intall BeautifulSoup`. It makes free-form queries of HTML and XML documents. It possesses a wide set of parsers that allow it to work with a variety of web XML–related formats. Some of these parsers are very strict, and some are very permissive.

```
from BeautifulSoup import BeautifulSoup
...
class TestBeautifulSoup(TestCase):

    def setUp(self):
        self.soup = BeautifulSoup(doc)

    def test_find_title_element(self):
        title = self.soup.find(name='title')[0]
        assert title.name == 'title'

    def test_find_body_by_attributes(self):
        body = self.soup.find(attrs={'bgcolor': '#ffffff'})
        assert body.name == 'body'
        assert body.attrs = [(u'bgcolor', u'#ffffff')]

    def test_find_by_text(self):
        # must match entire text string
        text = self.soup.find(text='Comic Feeds')
        assert text == 'Comic Feeds'
```

When the find arguments are used in the same expression, they are anded together, restricting the set of returned elements.

The name and text search attributes aren't limited to strings. They can be replaced by regular expressions. This is particularly useful in combination with the `findAll()` method, which returns all matches, rather than just the first one.

```
import re
...
def test_find_all_elements_with_e(self):
    has_e = self.soup.findAll(name=re.compile('e'))
    element_names = [x.name for x in has_e]
    assert element_names == ['head', 'title']
```

Testing JavaScript

Testing JavaScript is far more involved than testing other kinds of content. It poses many of the same problems as testing Python. As with Python, there are tools for performing both unit and functional tests. I'll only be dealing with the former in this chapter.

In order to unit test JavaScript, you have to be able to run the JavaScript code. There are stand-alone interpreters for JavaScript, but these have shortcomings compared with browsers. First and foremost, each browser has a slightly different set of libraries. Emulating these differences in a stand-alone interpreter is a technical challenge that nobody has risen to yet, nor are

they likely to. Since these changes must be tested, we're left with the option of executing the code with the target browsers.

This is done with the software package JsUnit (`www.jsunit.net/`), written by Edward Hieatt of Pivitol Labs. It is not to be confused with the similarly named JsUnit package written by Jörg Schaible.

Using JsUnit

The first step in working with JsUnit is obtaining a copy. It can be downloaded from `http://downloads.sourceforge.net/jsunit`. As of this time, there are two ZIP files available. One is an Eclipse plug-in, and the other is JsUnit itself. Sadly, the Eclipse plug-in does not work with the most recent versions of Eclipse (3.2 as of this writing), so I won't discuss its use.

We'll be using JsUnit in conjunction with the RSReader project from previous chapters. This is the first non-Python tool in the project, so it fits in a different place. I tend to create a generic tools directory when the project is small. Only when a particular class of tools gets large enough do I create dedicated directory hierarchies.

```
$ cd /Users/jeff/Documents/ws/rsreader
$ ls
```

```
build/                         setup.py
dist/                           setuptools-0.6c7-py2.5.egg
ez_setup.py                 src/
ez_setup.pyc                thirdparty/
setup.cfg
```

```
$ mkdir tools
$ cd tools
$ curl -L -o jsunit2.2alpha11.zip➥
    http://downloads.sourceforge.net/jsunit/jsunit2.2alpha11.zip
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 5968k  100 5968k    0     0   967k      0  0:00:06  0:00:06 --:--:-- 1169k
```

```
$ unzip jsunit2.2alpha11.zip
```

```
  inflating: jsunit/app/css/readme
  inflating: jsunit/app/emptyPage.html
...
  inflating: jsunit/tests/jsUnitUtilityTests.html
  inflating: jsunit/tests/jsUnitVersionCheckTests.html
```

```
$ ls
```

```
jsunit/                  jsunit2.2alpha11.zip
```

```
$ rm jsunit2.2alpha11.zip
```

The JavaScript source and tests will be placed in their own directory trees:

```
$ cd ..
$ mkdir javascript
$ ls
```

```
build/                          setup.py
dist/                           setuptools-0.6c7-py2.5.egg
ez_setup.py                  src/
ez_setup.pyc                 thirdparty/
javascript/                   tools/
setup.cfg
```

```
$ cd javascript
$ mkdir src
$ mkdir test
$ ls
```

```
src     tests
```

Running a Test

You can run tests stand-alone or distributed. Stand-alone tests are suitable for developing the tests themselves or interactively testing small pieces of code, as they require the user to interact with a web browser. Distributed tests are run from within the build. They use a farm of web browsers that may reside on other machines.

To start with, I'll demonstrate stand-alone testing. Once you've gained an understanding of how to use JsUnit, I'll move on to using distributed tests, in order to tie them into the larger build for automatic execution.

The JsUnit test runner is a web page in your browser. Open the browser of your choice to the file `rsreader/tools/jsunit/app/testRunner.html`. On my system, this is `file:///Users/jeff/Documents/ws/rsreader/tools/jsunit/testRunner.html`. The test runner is shown in Figure 10-1.

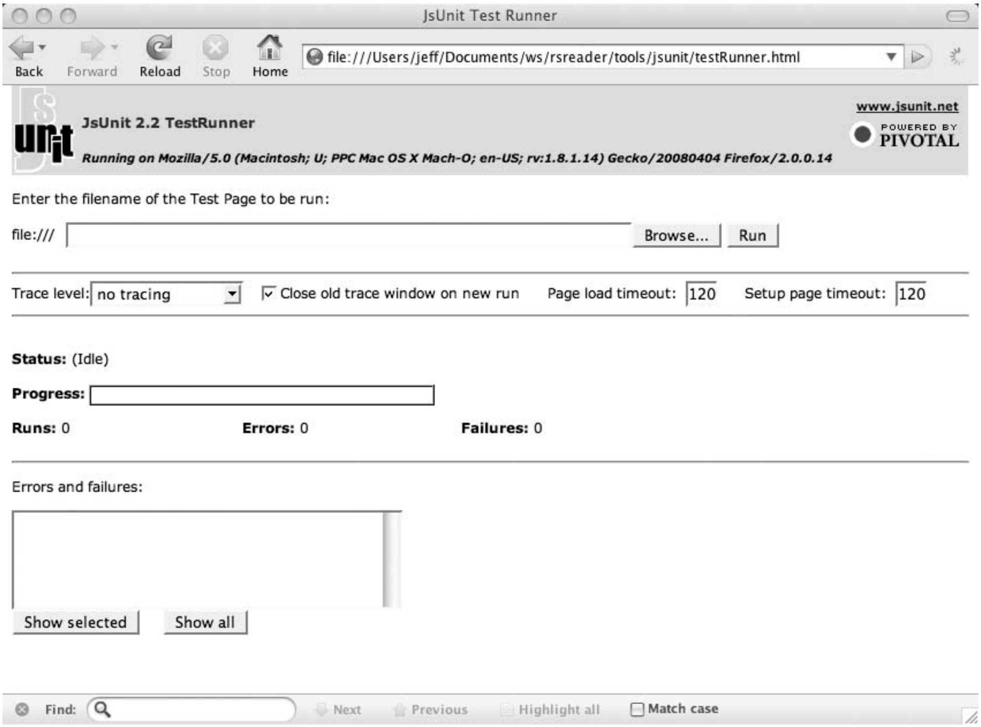


Figure 10-1. *The JsUnit stand-alone test runner*

The name of the file containing the tests to be run is put into the first text box. Clicking the Run button beside the text bar loads and runs the test.

A JsUnit test is an HTML file. JavaScript documents frequently manipulate the document structure, so it must be included as part of the test. HTML is the natural place to do this. The following file defines a function and a test for that function:

```
$ more test/lineTest.html
```

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript">

function line(m, x, b) {
    return m*x + b;
}
```

```
function testCalculationIsValid() {
    assertEquals("zero intercept", 10, line(5, 2, 0));
    assertEquals("zero slope", 5, line(0, 2, 5));
    assertEquals("at x = 10", 25, line(2, 10, 5));
}

        </script>
    </head>
    <body>
        This page tests line(m, x, b).
    </body>
</html>
```

First notice that all of the test code resides within the <head> tag. The first <script> tag is what makes this a JsUnit test:

```
<html>
...
        <script language="JavaScript"
                           src="../../tools/jsunit/app/jsUnitCore.js">
...
</html>
```

It loads all of the JsUnit test code that executes when the test page finishes loading. If anything goes wrong with this loading process, then you'll see the message shown in Figure 10-2.

Note I strongly advise you to adjust the "Page load timeout" and "Page setup timeout" to much smaller values. They are specified in seconds, and the defaults are 2 minutes (120 seconds). This is much too long when you're running tiny tests from the filesystem. Somewhere between 2 and 5 seconds is a reasonable value.

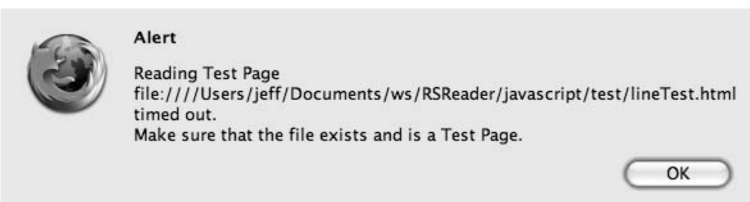


Figure 10-2. *The dreaded "Reading Test Page . . . timed out" alert*

The error in Figure 10-2 means one of two things. Either the file doesn't exist or the path to `jsUnitCore.js` is incorrect. You can check the former by trying to load the URL in a normal web browser. The latter is a bit trickier. Change to the test page's directory (in this case `/Users/jeff/Documents/ws/rsreader/javascript/test`), and then cut and paste the path in the script tag's `src` attribute.

This is the single most frustrating part of getting started with JsUnit.[3] The good news is that once it's ironed out, you won't have to deal with it again. If you're using Eclipse and Pydev, you should set up a template for these test pages so that nobody on your project will have to deal with this problem either.

A successful test run is shown in Figure 10-3. The progress bar is full and green, and the total test execution time is shown above it. Notice that this absolutely trivial test took over half a second to run.

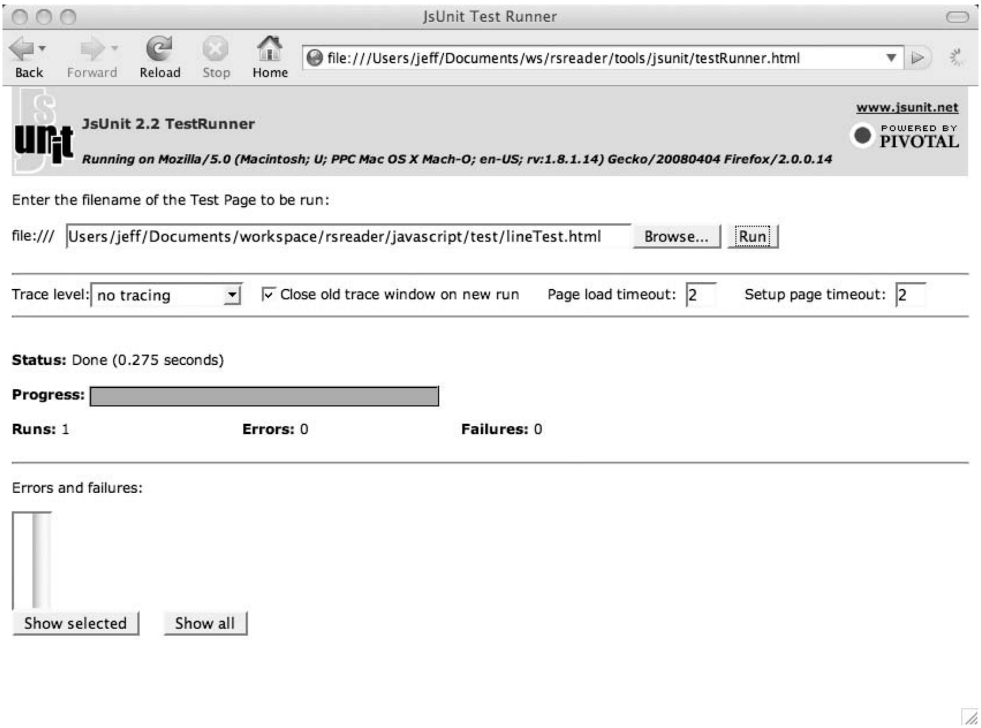


Figure 10-3. *The test runs and succeeds.*

JsUnit tests are *slow*. This highlights a theme with all JavaScript testing tools: write as few tests as possible. Don't do this by skimping on tests, though—do it by structuring your code so that you need to run as few tests as possible.

3. I suffer so you won't have to.

An excellent way of doing this is by depending on someone else to do the heavy lifting. There are wonderful JavaScript libraries available for free that implement the vast majority of things you'll want to do. MochiKit, script.aculo.us, and Ext JS are three of the most popular. Use them.

How It Works

Test cases are recognized by name. If a function begins with the string `test`, then it is treated as a test. JsUnit is a direct translation of the XUnit framework, as is Python's unittest. There is a one-to-one correspondence between most of the major concepts. This is shown in Table 10-3.

Table 10-3. *The Correspondence Between unittest and JsUnit*

unittest	JsUnit
`TestCase` classes	Test pages
Test methods	Test functions
Test suites	Test suite pages
Extend `unittest.TestCase`	Include `app/jsUnitCore.js`
Import subject code	Include subject code
`setUp()` and `tearDown()` methods	`setUp()` and `tearDown()` functions
IDE test runners	Web browser test runners

The correspondence between the two carries through to the test methods, too. These similarities are shown in Table 10-4.

Table 10-4. *The Correspondence Between unittest Test Methods and JsUnit Test Functions*

unittest Method	JsUnit Function
`assert_()`	`assert()`
`failUnless()`	`assertTrue()`
`assertTrue()`	`assertTrue()`
`failIf()`	`assertFalse()`
`assertFalse()`	`assertFalse()`
`assertEqual()`	`assertEquals()`
`failUnlessEqual()`	`assertEquals()`
`assertNotEqual()`	`assertNotEquals()`
`failIfEqual()`	`assertNotEquals()`
`failUnless(x is None)`	
`failIf(x is None)`	
`failUnless(x is None)`	`assertNull()`
`failIf(x is None)`	`asssertNotNull()`
	`assertNaN()`
	`assertNotNan()`

Connoisseur of the Undefined

If you're not familiar with JavaScript, there are a few methods that bear some explanation. Unlike Python, JavaScript draws a distinction between variable declaration and variable assignment.

In JavaScript, variables must be declared before they are used. Until you assign a value to that variable, its value is `undefined`. (It might have been clearer to call it *uninitialized*.) After you assign a value to the variable, it is no longer `undefined`. When you want to say that this variable has no value, you assign it the value `null`, which corresponds to Python's `None`.

As variables can be in two different indeterminate states, it is necessary to have two different sets of test methods. Since Python variables are created by the act of assignment, there is no such thing as a declared but unassigned variable, and there is no need for these test functions. The trade-off is that in Python, it is possible to create new variables accidentally by misspelling names. The following test function shows how `undefined` and `null` relate:

```
function testVariableInitializationStates() {
    var foo;
    assertUndefined(foo);
    assertNotNull(foo);

    foo = 0;
    assertNotUdefined(foo);
    assertNotNull(foo);

    foo = null;
    assertNotUndefined(foo);
    assertNull(foo);
}
```

The second feature visible in the test methods is the value `NaN`, which is short for *Not a Number*. JavaScript returns `NaN` from many arithmetic expressions that would raise exceptions in Python. Commonly, it also arises when a string-to-numeric conversion fails. The following Python test checks for just such a failure:

```
def testNumericConversionFailure(self):
    self.failUnlessRaises(ValueError, int, 'foo')
```

It is equivalent to this JavaScript test:

```
function testNumericConversionFailure() {
    assertNaN(parseInt('foo'));
}
```

The value `NaN` is a valid JavaScript number, and it can participate in normal computations. JavaScript also has special values to represent positive and negative infinities. In general, where Python will generate an exception such as `DivisionByZero`, JavaScript will return a sensible but not terribly useful symbol representing the numeric construct.

Python also has a `nan`, but it appears in fewer places. While Python mixes exceptions and special symbolic representations, JavaScript is pleasantly consistent in its usage.

Adding a Little More Realism

Note that in Table 10-3, I mentioned including subject code in the test. However, I didn't do that in the simple test example, and the subject function `slope(m, x, b)` is declared within the test itself. To make the example a bit more realistic, I'll move it to the source directory in a file named `line.js`, and I'll reference that from the test. The subject code is shown in Listing 10-1 and the test is shown in Listing 10-2.

Listing 10-1. *The Subject Code in rsreader/javascript/src/line.js*

```
function line(m, x, b) {
   return m*x + b;
}
```

Listing 10-2. *The Test Code in rsreader/javascript/test/lineTest.html*

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript" src="../src/line.js"></script>
        <script language="JavaScript">

function testCalculationIsValid() {
    assertEquals("zero intercept", 10, line(5, 2, 0));
    assertEquals("zero slope", 5, line(0, 2, 5));
    assertEquals("at x-axis", 25, line(2, 10, 5));
}

        </script>
    </head>
    <body>
        This a page tests line(m, x, b).
    </body>
</html>
```

With these changes in place, the test executes successfully, leaving you with a green bar in the test runner.

Manipulating the DOM

To do anything of interest, a JavaScript program must interact with the user, and to interact with the user, it must interact with the browser. This is done through the *Document Object Model (DOM)*. The DOM is a standard internal representation of the HTML document being presented to the user. It acts as the formal interface between the browser and the JavaScript interpreter.

Anything passing between the browser and your JavaScript code goes through the DOM. By embedding your tests within HTML pages, JsUnit gives them access to the DOM.

We're going to add a validation function to `line.js`. It will check an input field named `slope` in a form named `lineForm`. When there is an error, it will write an error message with a `<div>` tag with the ID `errorMsg`. The validation will be activated when someone clicks the `calculate` button. Listing 10-3 gives a skeleton test document that will be filled in as we walk through the process.

Listing 10-3. *The Test Skeleton for rsreader/javascript/test/testSlopeValidator.html*

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript" src="../src/line.js"></script>
        <script language="JavaScript">
           // tests go here
        </script>
    </head>
    <body>
         // DOM elements for the test go here
    </body>
</html>
```

The input will require a form named `slopeForm`. Since the form is not being submitted, we don't need to include an action.

```
<form name="lineForm">
</form>
```

The form needs to include the input text field named `slope`:

```
<form name="lineForm">
    <input type="text" name="slope"/>
</form>
```

The validator is activated when the user clicks the calculate button:

```
<form name="lineForm">
    <input type="text" name="slope"/>
    <input type="button" value="Calculate" onclick="validateSlope()"/>
</form>
```

The calculate button will never be called by the tests. It serves as documentation, showing how the tested method should be used within a real document.

The error messages will be presented in a `<div>` tag. When the validator is run, it will change the inner contents of this element. The tag is split in two to emphasize that it is the contents that are important.

```
<div id="errorMsg"></div>
<form name="lineForm">
    <input type="text" name="slope"/>
    <input type="button" value="Calculate" onclick="validateSlope()"/>
</form>
```

The location of the error message isn't important. It could just as well be part of the form, but placing it outside underscores this fact.

For the first tests, you'll just want to verify that the included subject page defines the method you want to test:

```
function testThatValidateSlopeIsDefinedAndIncluded() {
    validateSlope();
}
```

The tests will be looking at the value of the `<div>` tag, so it should be cleared to a known value before every test. This is done with a `setUp()` function:

```
function setUp() {
     document.getElementById('errorMsg').innerHTML = "";
}

function testThatValidateSlopeIsDefinedAndIncluded() {
    validateSlope();
}
```

This preceding JavaScript and the DOM fixtures are placed into the test skeleton. The minimal test is shown in Listing 10-4.

Listing 10-4. *A Minimal Test in rsreader/javascript/test/testSlopeValidator.js*

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript" src="../src/line.js"></script>
        <script language="JavaScript">

function setUp() {
     document.getElementById('errorMsg').innerHTML = "";
}

function testThatValidateSlopeIsDefinedAndIncluded() {
    validateSlope();
}

        </script>
    </head>
```

```
    <body>
        <div id="errorMsg"></div>
        <form name="lineForm">
            <input type="text" name="slope"/>
           <input type="button" value="Calculate" onclick="validateSlope()"/>
        </form>
    </body>
</html>
```

The `validateSlop()` function hasn't been defined yet, so when the test is run, you will see a failure, as shown in Figure 10-4.

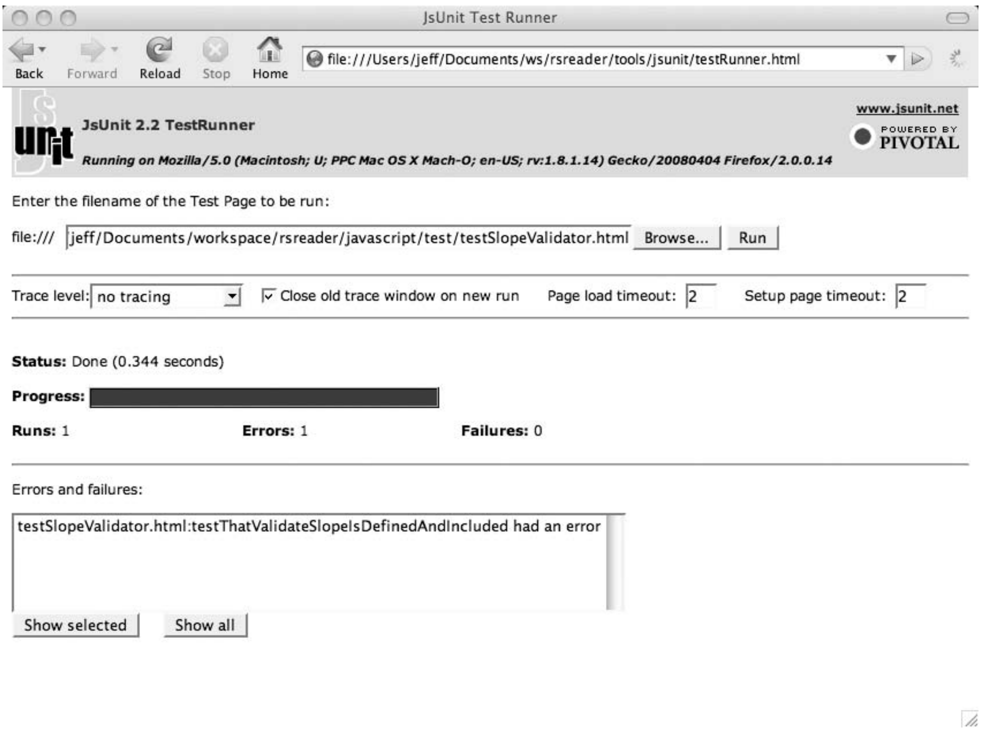


Figure 10-4. *The test dies because the subject hasn't been implemented.*

JsUnit, like unittest and Nose, distinguishes between failures and errors. Failures result from assertions failing, and errors result from the tests dying during execution. The combined failures and errors are shown in the "Errors and failures" panel.

Each line in the "Errors and failures" panel represents one unit test. Highlighting a test and clicking "Show selected" brings up the details for that test in an alert. Clicking "Show all" brings up a window with all the failure information. Failures will show detailed information about the failure, such as the expected value and the value produced. Errors will show a traceback. (If you've worked with JavaScript much, you'll be drooling at this prospect.) This is shown in Figure 10-5.

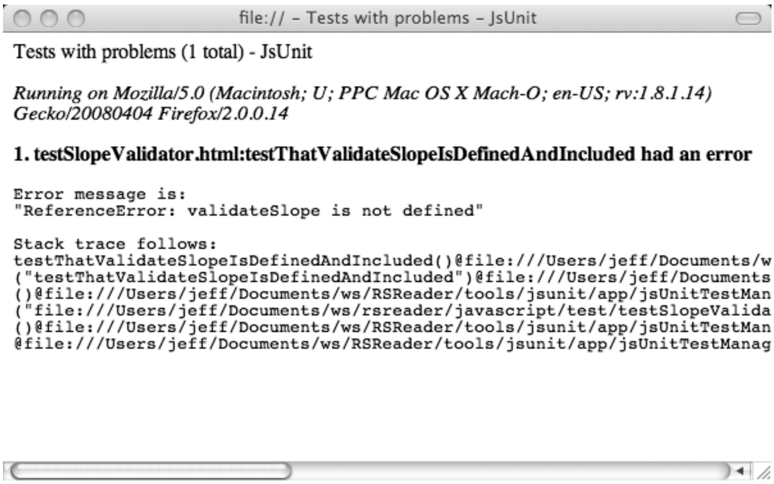


Figure 10-5. *The "Show all" errors window with a message and traceback*

Add a minimal definition to `line.js`. The file now reads as follows:

```
$ cat src/line.js
```

```
function validateSlope() {
}

function line(m, x, b) {
   return m*x + b;
}
```

Run the test again. This time it succeeds, as shown in Figure 10-6.

From this point forward, the process is very similar to developing with TDD in Python. You define a function that checks one kind of validation failure:

```
function testValidationFailsWhenEmpty() {
    document.lineForm.slope.value = '';
    validateSlope();
    var errorMsg = document.getElementById('errorMsg');
    assertEquals('You must define a slope', errorMsg.innerHTML);
}
```

This function demonstrates the classic XUnit test pattern. It sets the expectations, performs the action, and then checks the results, which are to be found within the <div id="errorMsg"> tag in the test page.



Figure 10-6. *The test runs successfully.*

You run the test and, as expected, it fails. You then add the corresponding logic to `validateSlope()`. The new definition is as follows:

```
function validateSlope() {
    var errorMsg = document.getElementById('errorMsg');
    errorMsg.innerHTML = "You must define a slope";
}
```

Now when you run the test, it succeeds. This back-and-forth process continues until the test cases are completed, as shown in Listings 10-5 and 10-6.

Listing 10-5. *The Subject Code in rsreader/javascript/src/line.js*

```
function line(m, x, b) {
   return m*x + b;
}

function validateSlope() {
    var slope  = document.lineForm.slope.value;
    var errorMsg = document.getElementById('errorMsg');
    if (!slope) {
        errorMsg.innerHTML = 'You must define a slope';
    } else if (isNaN(parseInt(slope))) {
        errorMsg.innerHTML = "The slope must be a number";
```

```
    } else {
        errorMsg.innerHTML = '';
    }
}
```

Listing 10-6. *The Test Code in rsreader/javascript/test/testSlopeValidator.html*

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript" src="../src/line.js"></script>
        <script language="JavaScript">

function setUp() {
    // clear out any previous message
    errorMsg().innerHTML = '';
}

function testFieldIsBlankAfterSuccessfulValidation() {
    document.lineForm.slope.value = '0';
    errorMsg().innerHTML = 'an arbitrary message';
    pressCalculate();
    assertEquals('', errorMsg().innerHTML);
}

function testValidationFailsWhenEmpty() {
    document.lineForm.slope.value = '';
    pressCalculate();
    assertEquals('You must define a slope', errorMsg().innerHTML);
}

function testValidationFailsWhenNotANumber() {
    document.lineForm.slope.value = 'this is not a number';
    pressCalculate();
    expected = 'The slope must be a number';
    assertEquals(expected, errorMsg().innerHTML);
}

// separate tests from validation mechanism details
function pressCalculate() {
    validateSlope();
}
```

```
// make tests more concise
function errorMsg() {
    return document.getElementById('errorMsg');
}

        </script>
    </head>
    <body>
       <form name="lineForm">
           <input type="text" name="slope" value="0"/>
           <input type="button" value="Calculate" onclick="validateSlope()"/>
           <div id="errorMsg"></div>
       </form>
    </body>
</html>
```

There are a few things worth pointing out about the final tests:

- The original test function `testThatValidateSlopeIsDefinedAndIncluded()` no longer serves any function, so it has been removed.
- The call to the subject method `validateSlope()` has been extracted into the method `pressCalculate()`. This makes it clearer what user action is being tested. It also makes the test less dependent on the name of the subject function. This is more of a worry with JavaScript, since we don't have refactoring tools at our disposal.
- The common subexpression `document.getElementById('errorMsg')` has been extracted into a utility method. This makes the test code more concise and moves this dependency into one place, reducing the test's brittleness.

Aggregating Tests

JsUnit tests can be combined using test suite pages. Test suite pages don't define any test functions. Instead they define a single function called `suite()`. This function, which you create, must return a `jsUnitTestSuite()` object. Test pages are added using the methods `addTestPage(testPagePath)` and `addTestSuite(testSuite)`. The argument to the latter must be a `jsUnitTestSuite`.

At this point, two tests have been defined. They are `testLine.js` and `testSlopeValidator.html`. These are combined into the suite page shown in Listing 10-7.

Listing 10-7. *The line.js Test Suite rsreader/javascript/test/lineSuite.html*

```
<html>
    <head>
        <title>Test Page line(m, x, b)</title>
        <script language="JavaScript"
                src="../../tools/jsunit/app/jsUnitCore.js">
        </script>
        <script language="JavaScript">
```

```
function suite() {
      var suite = new top.jsUnitTestSuite();
      suite.addTestPage("../../javascript/test/lineTest.html");
      suite.addTestPage("../../javascript/test/testSlopeValidator.html");
      return suite;
}
        </script>
    </head>
    <body>
        All tests for line.js.
    </body>
</html>
```

The trickiest part about making a test suite page is the getting the paths right. They are relative to `testRunner.html`, which in this example is in `rsreader/tools/jsunit/`.

Usually when you create test suites, you aggregate test pages or other suite pages. For the purposes of `addTestPage(testPagePath)`, there is no difference between a test page full of test functions and a test page that defines a suite. Accordingly, you will deal with test suite objects and `addTestSuite(testSuite)` infrequently. The following `suite()` function aggregates several test suite pages into a single suite:

```
function suite() {
    // the geometry suite
    var suite = new top.jsUnitTestSuite();
    suite.addTestPage("../../javascript/test/lineSuite.html");
    suite.addTestPage("../../javascript/test/circleSuite.html");
    return suite;
}
```

Running Tests by URL

Remember what I wrote about JsUnit being slow? The preceding test suite with four methods takes about 1.7 seconds on Firefox 2.0.0.14 on a dual-core 2.3 GHz processor.[4] Running an exhaustive set of JsUnit tests takes a long time. During development, you'll run tests frequently, so you'll want to break them up into meaningful and useful chunks that you can run as needed, and run quickly. Essentially, you'll use test suites as a means of optimizing your test runs. Frequently you'll run specific suites interactively.

You'll find yourself pasting, clicking, and running ad nauseam. Any means of making this process go faster will make your life easier and encourage you to use tests.[5] You can easily speed things up by supplying information as part of the test page URL.

---

4. In five years, we're both going to look back at this sentence and think, "Gee, that's not even as fast as the low-voltage CPU in my vacuum cleaner." I hate dating myself.

5. And that is what I want—I want you to test and test and test, simply because I like using software that isn't infested by bugs. I want you to write better software, and I want you to have more fun doing it, too.

Several parameters can be supplied as part of the test runner URL. The `testPage` parameter is the path to the test page that is run. The `autoRun` parameter tells the runner to immediately execute the test page. Instead of loading the test runner, typing in the test page, and then clicking Run, you could simply open the following URL to execute `lineSuite.html`:

```
file:///Users/jeff/Documents/ws/rsreader/tools/jsunit/testRunner.html?testPage=/➥
Users/jeff/Documents/ws/rsreader/javascript/test/lineSuite.html&autoRun=true
```

Values may also be passed to the test pages themselves. Arbitrary parameter/value pairs can be placed in the URL. When the page runs, these will be available to the tests through the object `top.jsUnitParmHash`. The parameters can be accessed as either `top.jsUnitParmHash.`*`parameterName`* or `top.jsUnitParmHash['`*`parameterName`*`']`.

Summary

The Web is a hodgepodge of rapidly evolving technologies, but at its core it is based on three things: a document format (HTML), a method of identifying documents (URIs/URLs), and a network protocol (HTTP) that retrieves those documents. Web browsers retrieve documents from web servers and present them to users. Those documents link to each other through embedded URIs, and browsers can follow those links.

HTML documents can act as forms. The user supplies the requested information, and it is sent back to a web server, which returns yet another document. Originally, all processing had to happen at the server, but this changed with the advent of JavaScript.

JavaScript is a powerful interpreted programming language that runs within the user's web browser. The programs are embedded within web pages; they can communicate across the network to the servers they were loaded from.

Today, most interesting web applications are based on web application frameworks of one sort or another. These offer the programmer a wide variety of services such as data persistence, HTML templating, and session management. Notable Python frameworks are Django, Google App Engine, Pylons, Turbogears, and Zope. These frameworks interface with the underlaying web servers using the Python protocol WSGI.

Web applications should be structured to facilitate unit testing. MVC is a common architectural choice that separates computation from output generation and input processing. In this way, each component can be tested independently of the others.

These components are often tested with different tools. Testing the computation or business logic is no different than with any other program. The input is also amenable to similar treatment. The real difference lies with the output. There are many specialized tools for dealing with the markup that most pages generate. Two useful Python libraries are `xml.etree.ElementTree` and `BeautifulSoup`.

JavaScript testing is the real challenge. It is a second programming environment, and it comes with its own tools. JavaScript programs are application code, so they should be developed with the same disciplines and practices as the rest of your programs. JavaScript implementations vary in important details from browser to browser, so it is necessary to test JavaScript within the target browsers. The most commonly used unit-testing tool is JsUnit. It operates in both stand-alone and distributed mode. Distributed mode has poor Python harness integration, so I only cover the stand-alone mode in this book.

Chapter 11 examines acceptance testing tools. These tools help to define the program's requirements, as well as ensure that the program behaves as expected. The tool I'll demonstrate is PyFit.

CHAPTER 11

Functional Testing

Functional testing is about building the right code. It is as important as unit testing, but it gets far less press. It breaks down into the three rough categories of *acceptance testing*, *integration testing*, and *performance testing*. I won't examine performance testing at all, and I'll only discuss integration testing in passing. This chapter's real meat is acceptance testing using PyFit, a functional and integration testing tool. So what is integration testing, and how do integration tests differ from acceptance tests?

Integration testing determines if large chunks of the application fit together correctly. It's like fitting together a few pieces of a broken mug before you try to glue the entire thing together. If you can't fit together the big chunks, then you know you can't reassemble it all. These sorts of tests are often not specified up front, but written by programmers or testers as the project proceeds.

Acceptance tests are begun before the program is written. In a perfect world, they serve as the outline for all the new features in an iteration of development. They are written in conjunction with the customer. Acceptance tests are an adjunct to stories. The stories are brief descriptions that provide a roadmap for the feature, but they don't supply anything concrete that can be automatically verified. That's where the tests come in.

The stories serve as a starting point for the discussion between the developers and the customer. These two hash out the details. The customer supplies the needed inputs and broad behaviors of the product. The customer comes from a high level, and the developer comes from a low level. Their goal is to meet in the middle in a place that captures the essence of the feature in way that the customer can understand, yet in enough detail that it can be quantified for testing. The product of this discussion is one or more acceptance tests.

Running Acceptance Tests

Acceptance tests occupy a different place in the build infrastructure than unit tests. The build fails if unit tests fails, but the product fails if acceptance tests fail. The build must always work, but the product doesn't have to work until delivery, so acceptance tests are not expected to pass with every build.

However, acceptance tests do yield useful information when run. Their successes and failures suggest how close the product is to completion. This information is interesting to developers in that it allows them to know how close they are to completion, but it's also interesting to customers. It should be available to both, and it should be produced regularly, but it doesn't need to be produced with every build.

The injunction against running functional tests with every build is even more important when you consider that functional tests are often slower than unit tests. Often they are orders of magnitude slower, in some instances taking literally days to run against mature products. Functional test farms are not unheard of with large products. Running them quickly can be a major engineering effort. At least one person I've spoken with has been porting their testing infrastructure to cloud computing environments such as Amazon's EC2 so that they can acquire hundreds of testing machines for short periods of time. Fortunately, I haven't had to confront such monsters myself.

This problem is remedied by adding a second kind of build to your continuous integration servers. The builds you've seen until now construct the software and then run the unit tests. I'll refer to them as *continuous builds*. The new builds do this, but they also run the functional tests after the unit tests complete. I'll refer to these as *formal builds*. Formal builds should run regularly, at least daily and preferably more often, and the results should be published to the customer.

PyFit

FIT (Framework for Integrated Tests, `http://fit.c2.com/`) is a tool developed by Ward Cunningham to facilitate collaboration between customers and developers. Tests are specified as tables, which are written in a tool the customer is familiar with. Developers or testers use these documents to write the tests. These tables are extracted from the documents, and they drive the acceptance tests. The end results are similarly formatted tables.

FITNESSE

FIT has given rise to a system called FitNesse, which is built around a wiki. Tests are entered as wiki pages, meaning that the people writing the tests need to learn a new tool, and they need to have access to the wiki in order to write tests. (Frankly, wikis are awful places to write tables.) Running the acceptance tests requires access to the wiki, too.

The real drawback for me is that the tests are independent of the code. It isn't possible to reproduce the acceptance criteria for a previous revision of the product. This may work for small groups or projects in which there is only ever one version of the system deployed at a time. While this is true of many hosted products, it's not true of many other software systems, particularly those that I work with.

FIT was originally produced for Java, but blessed clones have been created for many other languages. PyFit, written by John Roth, is Python's rendition. This flavor of FIT is well adapted to running from within the build. FIT has four components:

- *Requirement documents* are created by customers in conjunction with the developers. They specify the tests as tables, defining expected inputs and outputs, as well as identifying the associated test fixture.

- *Fixtures* are created by developers, and testers perform tests upon the applications.
- *Test runners* extract test data from tables in the requirement documents and then feed the data into the associated fixtures.
- *Reports* are created when the test runners are executed.

The relationship between these is shown in Figure 11-1.

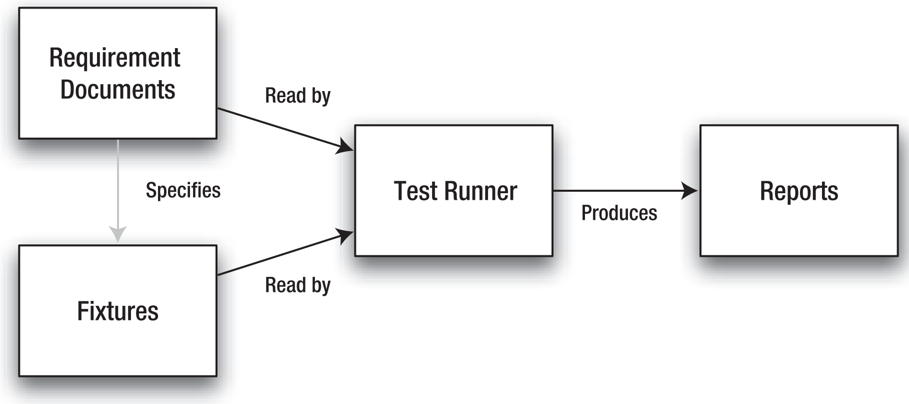


Figure 11-1. *The components of FIT*

These components are common across all FIT implementations. Requirement documents can come from the filesystem or from FitNesse servers (see the "FitNesse" sidebar). In this chapter, they'll all be coming from a local filesystem.

Writing Requirements

With some FIT implementations, the requirement documents can be in many different formats. The test tables can be extracted from Microsoft Word documents, Excel spreadsheets, and HTML documents. Any format capable of representing tables can theoretically be used as a source document as long as a converter is supplied.

My favorite document source is a spreadsheet. Spreadsheets are eminently capable of creating, manipulating, and formatting tables, and everyone knows how to use one. In particular, the customers I work with have extensive experience with them.

Now for the bad news: PyFit doesn't support them. With PyFit, you get one choice: HTML files. The good news is that there are many tools that will edit HTML files so that you don't have to get your hands dirty. Figure 11-2 is a FIT spec being written with Microsoft Word.

Geometry line calculation
Lines are a basic geometric construction, and we need to handle them. We don't need to do more than calculate the domain o f a single point given the range, as this will suffice for the graphing package.

Inputs

slope	slope of the line
x	supplied domain
intercept	intercept of the line
y?	calculated range

slope	x	intercept	y?
5	3	0	15
0	3	2	2

Figure 11-2. *Writing an HTML spec document using Microsoft Word*

First and foremost, nonprogrammers are the intended audience for this document. It has a format that you'll use again and again:

- The first section describes the purpose of the acceptance test in human terms. It gives a background for everything that follows.
- The second section describes the variables used in the test. On one side are the names, and on the other are descriptions. Variable names ending in ? are results calculated by the test fixture.
- One or more tables follow. These define the acceptance criteria.

Notes may be freely mixed within the document. The documents may be as simple or as fancy as you desire. When the FIT runner processes the document, it will extract the tables and use them to drive the fixtures.

These specifications are not intended to be data-driven tests that exhaustively examine every possible input and output. The rows should specify interesting conditions. This data should emphasize the things that are important to determine about the test. It is a waste of everyone's time to supply 50 or 60 rows when only a few are necessary to convey a complete explanation of how the feature is supposed to work.

So who is everyone? Everyone includes the customers, the developers, and the testers. The preceding spec would most likely be created by the customer. It's rough and it needs refinement. The document might be shuttled back and forth by e-mail a few times while people discuss the possibilities. Eventually, the team huddles around someone's laptop and hashes out a finished version. The precise process by which this happens isn't important.

What matters is that a discussion happens between all stakeholders. This requirement document serves as the centerpiece for discussion. It forces everyone to decide on a concrete description of the feature.

Along the way, the team creates a common vocabulary describing the application and its actions. This vocabulary defines the system metaphor. At first, this vocabulary grows quickly, but the birth rate of new terms declines quickly.

Because the group creates these documents, they are at a level that all parties can understand. Each party involved will pull the documents in their own directions. The customers will want them to be too abstract, and the developers will want them to be too concrete. It will take a while before the participants learn to choose the right level. This is a good time to use people with both customer- and application-facing experience, such as sales engineers. They can serve as arbitrators early in the process.

This is FIT's magic. The documents are abstract enough that nontechnical people can grasp them and learn to write them with familiar tools, and they're detailed enough to produce tests from. Their level of abstraction allows them to serve as design specifications, and their level of detail makes them sufficient to replace technical requirement documents and test plans. Since they can be executed, they serve as formal acceptance criteria, which can be verified through automation. This also means that they won't fall out of date.

There is one crucial thing missing from the specification in Figure 11-2. FIT has no way of knowing which test fixture to use. This information is added to the form by the developer when they begin writing the test implementation. The new table is shown in Figure 11-3.

geometry.line.CheckCoordinates			
slope	**x**	**intercept**	**y?**
5	3	0	15
0	3	2	2

Figure 11-3. *A fixture binding has been added to the table.*

In the figure, a fixture binding has been added to the beginning of the table. Customers know that this line is techie magic stuff, so they avoid modifying it.

The first row in Figure 11-3 binds the fixture `geometry.line.CheckCoordinates` to the values in the table. A developer or tester creates the fixture. This is a simple (but broken) example.

```
from fit.ColumnFixture import ColumnFixture
class CheckCoordinates(ColumnFixture):
    _typeDict={
        "slope": "Float",
        "x": "Float",
        "intercept": "Float",
        "y": "Float",
        }
    slope = 0.0
    x = 0.0
    intercept = 0.0
    def y(self):
        return self.slope * self.x
```

When FIT runs, it produces an XML summary document and an HTML page for every test. The HTML page for this test and fixture is shown in Figure 11-4.

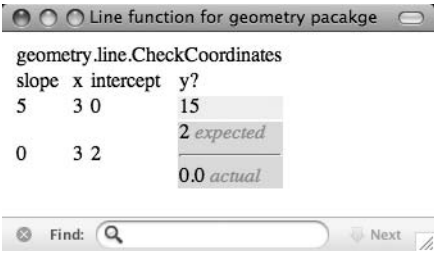


Figure 11-4. *A FIT report for a broken test*

Successful test results are shown in green and unsuccessful ones are shown in red.

A link to these documents may be supplied to the customers and to management. This provides them with a self-service view into the status of the current development iteration. Once an organization adapts to using FIT as its primary specification tool, these reports supplant many status meetings and other formal communications.

At the beginning, customers and management will have to become familiar with the rhythms and patterns with which features are fulfilled. This will take time, and that needs to be made up front. Care should be taken when introducing the process, and expectations should be managed carefully.

Once everyone is comfortable with the process, it has social benefits. Developers and managers will feel less need to pester developers for project statuses. Developers will feel less harried and less pressured, and it will give them a greater sense of control.

It will also give customers and managers the feeling of more control, too. Instead of harrying development (which management likes doing as little as development likes receiving), they can look to the day's reports. Regular meaningful feedback that they can retrieve empowers them, and it builds trust in their team.

A Simple PyFit Example

The previous section has hopefully given you a good feeling for what FIT does and how it can benefit you. Unfortunately, setting up FIT is more complicated than it needs to be. The documentation is patchy at best, and if you're not using FitNesse, it fails to address many implementation questions, particularly to do with running PyFit from a build. Fortunately, this lack of information conceals a simple process.

It should be easy to set up a build on a new machine. Each additional package that you have to install to perform a build is a potential barrier. Each step is another delay when new people start on the project, when a machine is rebuilt, or when a new build server is added. Each new package has to be back-ported to all the existing build environments, too. There is little as frustrating as updating your source and discovering that you need to install a new package, so your build should carry its own infrastructure whenever possible.

The best way to learn about PyFit is to work with it. You can install it via `easy_install`, but none of the executables will work, and it won't be accessible to the build. Instead you'll install it into the `tools` directory that was created for JsUnit in Chapter 10, and the build will run it from there.

As I write, the current version is 0.8a2, and you can download it directly from `http://pypi.python.org/packages/source/P/PyFIT/PyFIT-0.8a2.zip`.

Note As of this writing, an earlier version is also available from the Download Now section of the FIT web site, at `http://fit.c2.com`. Hopefully, it will be up to date by the time you read this.

```
$ curl -o /tmp/PyFIT-0.8a2.zip -L➥
http://pypi.python.org/packages/source/P/PyFIT/PyFIT-0.8a2.zip
```

```
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                Dload  Upload   Total   Spent    Left  Speed
100  962k  100  962k    0     0   314k      0  0:00:03  0:00:03 --:--:--  356k
```

```
$ cd /Users/jeff/Documents/ws/rsreader/tools
$ ls -F
```

```
jsunit/
```

```
$ unzip /tmp/PyFIT-0.8a2.zip
```

```
Archive:  /tmp/PyFIT-0.8a2.zip
  inflating: PyFIT-0.8a2/PKG-INFO
  inflating: PyFIT-0.8a2/README.txt
  ...
  inflating: PyFIT-0.8a2/fit/tests/VariationsTest.py
  inflating: PyFIT-0.8a2/fit/tests/__init__.py
```

```
$ ls -F
```

```
PyFIT-0.8a2/     jsunit/
```

The tests will always run with the version of PyFit in `tools`, so the version information in the file name is superfluous.

```
$ mv PyFIT-0.8a2 pyfit
$ ls -F
```

```
pyfit/        jsunit/
```

Finally, you can remove the ZIP file that you downloaded earlier:

```
$ rm /tmp/PyFIT-0.8a2.zip
```

At this point, you should check the `pyfit` directory and all of its contents.

Giving the Acceptance Tests a Home

Unlike unit tests, acceptance tests do not run every time the code is built. They are run when the developer needs to see the results, or when iteration progress is checked. The latter is typically done on a regular basis by a special build. This means that the acceptance tests must be separated from unit tests. You can do this by creating a directory for acceptance tests:

```
$ cd /Users/jeff/Documents/ws/rsreader
$ mkdir acceptance
$ ls -F
```

```
acceptance/    ez_setup.py      setuptools-0.6c7-py2.5.egg
build/           javascript/         src/
dist/            setup.cfg           thirdparty/
ez_setup.py      setup.py            tools/
```

You must have locations to store requirement documents, fixtures, and reports, and they should be separate:

```
$ mkdir acceptance/requirements
$ mkdir acceptance/fixtures
$ mkdir acceptance/reports
```

You should check these into your source repository at this point.

Your First FIT

Requirement documents are at the heart of FIT. There are a number of different families of tests that can be created with FIT. The type of test I'm showing you how to create is a column fixture. A *column fixture* is a table in which each column represents a different input or output to the test. Each row is a different combination of these values.

You're limited to HTML documents at this time—it's the format that PyFit currently understands. This doesn't mean that you have to write them by hand, though. Microsoft Word, Adobe Dreamweaver, or any tool capable of reading and writing HTML will speed the job along. The requirement document shown following is written to the file `acceptance/requirements/geometry/line.html`:

```
$ cat acceptance/requirements/geometry/line.html
```

```
<html>
    <head>
        <title>Line function for geometry pacakge</title>
    </head>
    <body>
        <table>
             <tr><td colspan="4">geometry.line.CheckCoordinate</td></tr>
             <tr>
                    <td>slope</td>
                    <td>x</td>
```

```
                <td>intercept</td>
                <td>y?</td>
            </tr>
            <tr>
                <td>5</td>
                <td>3</td>
                <td>0</td>
                <td>15</td>
            </tr>
            <tr>
                <td>0</td>
                <td>3</td>
                <td>2</td>
                <td>2</td>
            </tr>
        </table>
    </body>
</html>
```

The test-specific information is printed in bold. As with all HTML, it's much easier to interpret in a browser, as Figure 11-5 shows.

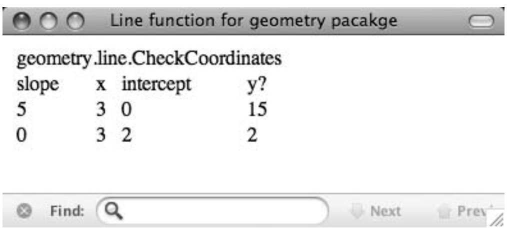


Figure 11-5. *The simple test table shown in a browser*

This table represents column fixture, and it has three important parts:

- The first row has one cell that reads `geometry.line.CheckCoordinates`. This designates the fixture for the test.
- The second row assigns variable names to columns. Columns names ending in ? are expected results from the tests.
- Each subsequent row contains a set of test values. A test fixture is constructed for each one, and the values from each column are used to either prepare the test or compare the results.

Column fixtures are commonly used to represent business process. A friend's favorite FIT examples come from a marketing department that he worked in. The tables consisted of criteria for when coupons should be given to customers. The conditions were involved, but walking a person through the process answered many questions about which values were important and which were not. They were all easily represented as columns.

There are other kinds of fixtures, and they have different behaviors, but they're all specified as tables.

The Fixture

A fixture can be thought of as a kind of command object. A fixture is created, the test values are set in the fixture, and then command methods are called. The variable names and command method names match those in the table.

Here, the fixtures are placed into `acceptance/fixtures`, which is the root of a Python package tree. Since it is a Python package tree, each subdirectory needs to have an `__init__.py` file.

```
$ mkdir acceptance/fixtures/geometry
$ touch acceptance/fixtures/geometry/__init__.py
$ mkdir acceptance/fixtures/geometry/line
$ touch acceptance/fixture/geometry/line/__init__.py
```

You'll notice something weird here—the binding line in the table is `geometry.line.CheckCoordinates`. It would be reasonable for you assume that it specifies the class `CheckCoordinates` in the module `geometry.line`, but you would be wrong. It actually specifies the class `CheckCoordinates` in the module `geometry.line.CheckCoordinates`. Go figure. Here is the fixture's code:

```
$ cat acceptance/fixtures/geometry/line/CheckCoordinates.py
```

```
from fit.ColumnFixture import ColumnFixture

class CheckCoordinates(ColumnFixture):
    _typeDict={
        "slope": "Float",
        "x": "Float",
        "intercept": "Float",
        "y": "Float",
        }

    slope = 0.0
    x = 0.0
    intercept = 0.0

    def y(self):
        return self.slope * self.x + self.intercept
```

The fixture breaks into three parts. All of the names correspond to columns in the requirements table minus any meaningful punctuation, such as the trailing ? on test results.

- The first section declares the test's variables and their types. This reflects FIT's strongly typed Java heritage. The argument types could be inferred from member definitions.
- The second section declares and initializes defaults for the variables that FIT supplies from each row in the requirements table.
- The third section defines the methods that produce results. Each is named after the corresponding column in the requirements table.

For this fixture, you can think of the execution process as following these steps:

```
f = CheckCoordinates()
converted_row = converted_values(row, f._typeDict)
f.slope = converted_row['slope']
f.x = converted_row['x']
f.intercept = converted_row['intercept']
f.recordAssertionResults('y?', converted_row['y?'], f.y())
```

The reality is more complicated, but this captures the essence of the process.

Running PyFit

PyFit supplies many different programs for running tests. These are located in `tools/pyfit/fit`. You want `FolderRunner.py`, which reads requirement documents from one directory and writes the finished reports to another. `FolderRunner.py` came from a ZIP file, and on UNIX systems this means that the execute bit isn't set, so you'll have to call Python to run it.

```
$ python tools/pyfit/fit/FolderRunner.py acceptance/requirements acceptance/reports
```

```
Result print level in effect. Files: 'e' Summaries: 't'
Total tests Processed: 0 right, 0 wrong, 0 ignored, 0 exceptions
```

This indicates that no tests were found. By default, `FolderRunner.py` searches only the top-level directory you specified, but `line.html` is in a subdirectory. The `+r` flag tells `FolderRunner.py` to search for tests recursively.

```
 $ python tools/pyfit/fit/FolderRunner.py +r acceptance/requirements➥
acceptance/reports
```

```
Result print level in effect. Files: 'e' Summaries: 'f'
Processing Directory: geometry
0 right, 0 wrong, 0 ignored, 1 exceptions line.html
Total this Directory: 0 right, 0 wrong, 0 ignored, 1 exceptions
Total tests Processed: 0 right, 0 wrong, 0 ignored, 1 exceptions
```

This is better. It found the test—however, it should display `1 right, 0 wrong, 0 ignored, 0 exceptions`. Obviously there's a problem here, so you'll want to look at the results. The test was in `acceptance/requirements/geometry/line.html`, so the results are in `acceptance/reports/geometry/line.html`.

```
$ cat acceptance/reports/geometry/line.html
```

```
...
            <tr><td colspan="4" class="fit_error">geometry.line.CheckCoordinates➥
<hr>The module 'geometry.line' was not found.</td></tr>
...
```

The relevant message in the preceding report is shown in bold. The module `geometry.line` wasn't found because the directory `acceptance/fixtures` isn't on the `PYTHONPATH`. This is something that can be easily fixed:

```
$ export PYTHONPATH=acceptance/fixtures
$ python tools/pyfit/fit/FolderRunner.py +r acceptance/requirements➥
acceptance/reports
```

```
Result print level in effect. Files: 'e' Summaries: 'f'
Total tests Processed: 1 right, 0 wrong, 0 ignored, 0 exceptions
```

This is exactly what you wanted. The tests have run and the results have been generated.

Making It Easier

You're going run PyFit frequently. Remembering all those values is a hassle, and you're certainly not going to want to type them over and over again, so you should create a script to do it for you.

The tool is going to have to go someplace, and you don't have a location for generic tool scripts yet. You can put them in a subdirectory of `tools`:

```
$ mkdir tools/bin
```

The script is called `tools/bin/accept.py`. The first version is shown in Listing 11-1.

Listing 11-1. *The First Pass at the Acceptance Testing Script*

```
#!/usr/local/bin/python

from subprocess import Popen
import sys

cmd = "%(python)s %(fitrunner)s +r %(requirements)s %(reports)s"
expansions = dict(python=sys.executable,
                  fitrunner='./tools/pyfit/fit/FolderRunner.py',
                  requirements='./acceptance/requirements',
                  reports='./acceptance/reports')
env = dict(PYTHONPATH='acceptance/fixtures')

proc = Popen(cmd % expansions, shell=True, env=env)
proc.wait()
```

On UNIX systems, you must make the script executable before it can be run:

```
$ chmod a+x tools/bin/accept.py
```

Before you can verify that it's doing the right thing, you need to remove PYTHONPATH from your shell's environment:

```
$ unset PYTHONPATH
```

At this point, running FolderRunner.py by hand results in an exception again:

```
$ python tools/pyfit/fit/FolderRunner.py +r acceptance/requirements➥
acceptance/reports
```

```
Result print level in effect. Files: 'e' Summaries: 'f'
Processing Directory: geometry
0 right, 0 wrong, 0 ignored, 1 exceptions line.html
Total this Directory: 0 right, 0 wrong, 0 ignored, 1 exceptions
Total tests Processed: 0 right, 0 wrong, 0 ignored, 1 exceptions
```

However, running the script works:

```
$ tools/bin/accept.py
```

```
Result print level in effect. Files: 'e' Summaries: 'f'
Total tests Processed: 1 right, 0 wrong, 0 ignored, 0 exceptions
```

There is still a problem, though. The script will only run from the project's root directory:

```
$ cd tools
$ bin/accept.py
```

```
/Library/Frameworks/Python.framework/Versions/2.5/Resources/Python.app/Contents/➥
MacOS/Python: can't open file './tools/pyfit/fit/FolderRunner.py':➥
[Errno 2] No such file or directory
```

All the paths are relative, which is good. It means that the tools work no matter where the project is placed on the filesystem. It's bad, however, in that the script will only run from the project's root directory. You could require people to run it from there, but that's inconvenient—people forget rules like that, so this isn't a particularly robust solution.

You could extract the root directory from an environment variable and require people to set that, but this creates problems for people working on multiple branches. In order to move between them, they'll have to reconfigure their environment.

A more robust solution determines the directory from the path. The revised program is shown in Listing 11-2.

Listing 11-2. *The Script accept.py Now Runs from Any Directory Within the Project*

```
#!/usr/local/bin/python

import os
from subprocess import Popen
import sys

def bin_dir():
    return os.path.dirname(os.path.abspath(__file__))

def find_dev_root(d):
    setup_py = os.path.join(d, 'setup.py')
    if os.path.exists(setup_py):
       return d
    parent = os.path.dirname(d)
    if parent == d:
       return None
    return find_dev_root(parent)

dev_root = find_dev_root(bin_dir())
if dev_root is None:
     msg = "Could not find development environment root"
     print >> sys.stderr, msg
     sys.exit(1)
os.chdir(dev_root)

cmd = "%(python)s %(fitrunner)s +r %(requirements)s %(reports)s"
expansions = dict(python=sys.executable,
                  fitrunner='./tools/pyfit/fit/FolderRunner.py',
                  requirements='./acceptance/requirements',
                  reports='./acceptance/reports')
env = dict(PYTHONPATH='tools/pyfit/fit:acceptance/fixtures')

proc = Popen(cmd % expansions, shell=True, env=env)
proc.wait()
```

The project's root is the ancestor of the `tools/bin` directory that contains `setup.py`. The functions `bin_dir()` and `find_dev_root(directory)` perform this search. The solution is a mouthful, but it can be used over and over again. When you need to reuse it, you should move it into a common module that gets installed with your development environment.

The script now runs from anywhere:

```
$ cd /tmp
$ /Users/jeff/Documents/ws/rsreader/tools/bin/accept.py
```

```
Result print level in effect. Files: 'e' Summaries: 'f'
Total tests Processed: 1 right, 0 wrong, 0 ignored, 0 exceptions
```

Besides learning a general technique, you've made the job of running from Buildbot that much easier.

FIT into Buildbot

In a world with limitless computing resources, there would only be one kind of a build. That build would execute the program's full test suite. It would run every unit test, acceptance test, functional test, and performance test. In the real world, however, there are rarely enough resources to do this.

Large mature projects often have full test suites that take hours if not days to run. Many of the tests thirstily consume resources. For obvious reasons, performance tests are consummate gluttons. Functional tests for products such as embedded systems or device drivers may require specialized hardware.

While the size of the unit test suite is roughly proportional to the size of the code base, the size of the functional test suite is proportional to the code base's age. As the code ages, the functional suite bloats. Eventually, it may grow so large that massive parallelization is the only solution to running it in a reasonable time.

Functional tests need to be broken out long before then, and acceptance tests do, too. The team needs regular progress reports for the acceptance tests, so the build containing them is produced at regular intervals.

Casting back to Chapter 5, you'll recall that we set up a build master and a build slave named `rsreader-linux`. The build system produced builds for both Python 2.4 and Python 2.5. The builds were triggered whenever a change was submitted to the Subversion server.

You're about to add a second kind of build that includes the acceptance tests. This build will run several times a day, and it will run even if there are no recent changes.

You do this by defining the following items in the Buildbot configuration file `master.cfg`:

- A *schedule* that determines when a *builder* runs
- A *build factory* that constructs a build
- *Build steps* that the build performs
- A *builder* that ties together a build factory, a slave, and a factory

Before you can run a builder, you need to set up the infrastructure on the slave.

Preparing the Slave

Each builder needs a unique build directory under `/usr/local/buildbot/slave/rsreader`. The previous two were `full-py2.4` and `full-py2.5`. This will be a Python 2.5 builder, and you should name it `acceptance-py2.5`.

```
slave$ cd /usr/local/buildbot/rsreader/slave
slave$ sudo -u build mkdir acceptance-py2.5
```

The builder needs its own Python installation. You can copy that from `full-2.5`.

```
slave$ sudo -u build cp -rp full-py2.5/python2.5 acceptance-py2.5/python2.5
```

Now you can set up the builder.

Run New Builder, Run!

There's an old maxim about software: Make it run. Make it run right. Make it run fast. It applies to configuration, too. Right now, you want to focus on making the builder run. Later, you can make it run right. This prevents you from conflating basic configuration errors with the mistakes you make while hacking out a new builder, so you should configure the build like the existing ones. They are defined by `/usr/local/buildbot/master/rsreader/master.cfg` as follows:

```
b1 = {'name': "buildbot-full-py2.5",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.5",
      'factory': pythonBuilder('2.5'),
      }
b2 = {'name': "buildbot-full-py2.4",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.4",
      'factory': pythonBuilder('2.4'),
      }
c['builders'] = [b1, b2,]
```

Adding the new definition gives you this:

```
...
b2 = {'name': "buildbot-full-py2.4",
      'slavename': "rsreader-linux",
      'builddir': "full-py2.4",
      'factory': pythonBuilder('2.4'),
      }
b3 = {'name': "buildbot-acceptance-py2.5",
      'slavename': "rsreader-linux",
      'builddir': "acceptance-py2.5",
      'factory': pythonBuilder('2.5'),
      }
c['builders'] = [b1, b2, b3]
```

After doing this, you should reload the Buildbot configuration to see if you've introduced any errors:

```
master$ buildbot reconfig /usr/local/buildbot/master/rsreader
```

```
sending SIGHUP to process 52711
2008-05-05 15:59:56-0700 [-] loading configuration from /usr/local/buildbot/master
2008-05-05 15:59:56-0700 [-] updating builder buildbot-full-py2.4: factory changed
...
Reconfiguration appears to have completed successfully.
```

The reconfig worked, so you can safely continue. The new builder must be scheduled. Here is the section of `master.cfg` containing the new scheduler definition:

```
####### SCHEDULERS

from buildbot.scheduler import Nightly, Scheduler
c['schedulers'] = []
c['schedulers'].append(Scheduler(name="rsreader under python 2.5",
                                 branch=None,
                                 treeStableTimer=60,
                                 builderNames=["buildbot-full-py2.5"]))
c['schedulers'].append(Scheduler(name="rsreader under python 2.4",
                                 branch=None,
                                 treeStableTimer=60,
                                 builderNames=["buildbot-full-py2.4"]))
c['schedulers'].append(Scheduler(name="Acceptance tests under python 2.5",
                                 branch=None,
                                 treeStableTimer=60,
                                 builderNames=["buildbot-acceptance-py2.5"]))
```

At this point, you should reconfigure the master. A quick look at the waterfall display in Figure 11-6 shows that the builder is online.

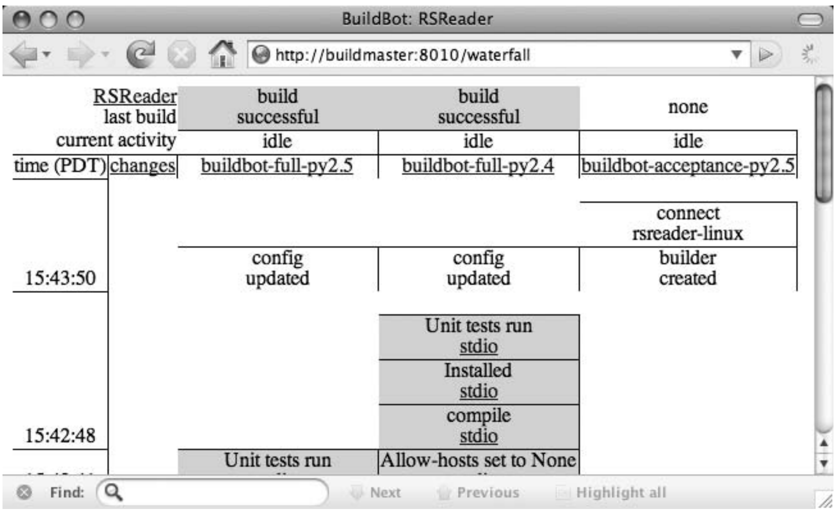


Figure 11-6. *The new builder is alive.*

You send a notification to verify that the builder actually works:

```
master$ buildbot sendchange --master buildmaster:4484 -u jeff -n 30 setup.py
```

```
change sent successfully
```

You can watch the build happen on the waterfall display.

Figure 11-7 shows the build completing on my system. You should see something similar. Now that you know the builder runs, you have to make it run right. Running the acceptance tests requires making a new builder factory.

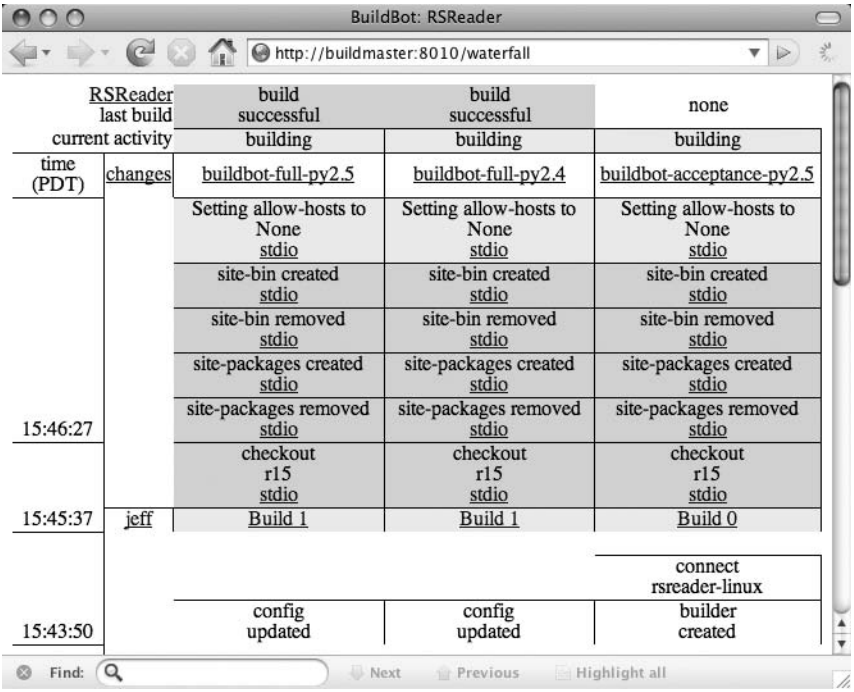


Figure 11-7. *The new builder builds.*

The current builder factory does most of what you want. You can easily leverage this. The new builder will call the old one and then add its own steps.

```
def pythonBuilder(version):
    python = python_(version)
    nosetests = nosetests_(version)
    site_bin = site_bin_(version)
    site_pkgs = site_pkgs_(version)

    f = factory.BuildFactory()
...
    f.addStep(ShellCommand,
              command=[python, "./setup.py", "test"],
              description="Running unit tests",
              descriptionDone="Unit tests run")
    return f
```

```
def pythonAcceptanceBuilder(version):
    f = pythonBuilder(version)
    f.addStep(ShellCommand,
              command=[python_(version, "./tools/bin/accept.py"],
              description="Running acceptance tests",
              descriptionDone="Acceptance have been run")
    return f
```

You should reload the master to ensure that you haven't made any mistakes. The new builder definition still references the old builder factory, so you make the following change to hook in the new one:

```
b3 = {'name': "buildbot-acceptance-py2.5",
      'slavename': "rsreader-linux",
      'builddir': "acceptance-py2.5",
      'factory': pythonAcceptanceBuilder('2.5'),
     }
```

Now you should reload the master again and verify that you didn't make an error. With that verified, you should trigger a build again:

```
master$ buildbot sendchange --master buildmaster:4484 -u jeff -n 30 setup.py
```

```
change sent successfully
```

When the build completes, you'll see the new step, as shown in Figure 11-8.

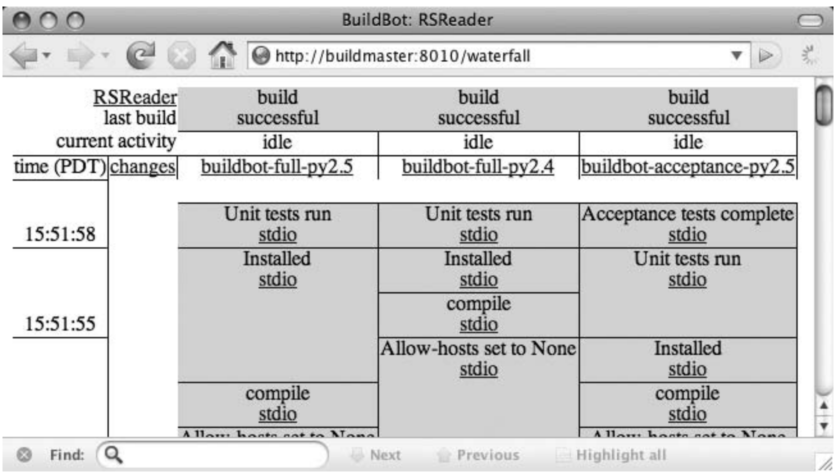


Figure 11-8. *PyFit has been run.*

The output from the build step shows that the build has been run successfully, but reports aren't available.

Making the Reports Available

When the PyFit step completes, the results are left in `acceptance/reports`. They're HTML documents, so you can publish them by copying them into a web server's document tree. Since you'll invariably have more than one build slave, you'll need to centralize the reporting.

Fortunately, this is easy to do with Buildbot. It can copy files from the slaves to the master, and you can use the master's internal web server to present the reports.

You'll want to track the project's progress over time, so you'll want to publish all the acceptance reports simultaneously. Copying them to a fixed location on the master won't work. If you do that, then the latest reports will overwrite the previous reports, so you'll need to copy each to a different location.

The combination of the Python version and Subversion revision make a useful identifier. These should effectively be unique. You'll copy the results to

```
public_html/rsreader/acceptance/%(subversion_revision)s-py%(python_version)s/reports
```

Buildbot makes build-related information such as the branch and the revision available through build parameters. The documentation would lead you to believe that the feature is complete, but it isn't.[1] The mechanisms for setting and reading build properties are in place, but the final pieces that set them are missing. Along the way, you're going to have to supply some of this machinery yourself.

The publishing process has the following steps:

- The slave packages the reports.
- The master copies the reports from the slave.
- The master unpacks and moves the reports to the web server.

Each of these corresponds to a build step.

Packaging the Reports

The build tool `accept.py` puts the reports into `acceptance/reports`. These files need to be zipped into a single archive so that the next step can copy them over. At this point, only this directory must be copied, but that is likely to change in the future. It's temping to issue shell commands directly, and this will work. However, to do this, Buildbot must understand how the build is structured.

Using formal interfaces allows developers to restructure the build without having to modify the build server. A red flag should go up when you find yourself tweaking the build system to manipulate build's internal structure. Instead you resolve the issue by putting the details within the build itself; this often involves creating some sort of tool.

You're about to create a tool called `package_reports.py`. Like `access.py`, it lives in `rsreader/tools/bin`. Here's the code:

1. This is true as of Buildbot 0.7.7.

```
#!/usr/local/bin/python

import os
from subprocess import Popen
import sys

if len(sys.argv) != 2:
    msg = """Usage: %s ZIPFILE

This must be run from the project's root.
"""
    print >> sys.stderr, msg % sys.argv[0]
    sys.exit(0)

filename = sys.argv[1]
reports = os.path.join("acceptance", "reports")
proc = Popen(['zip', '-r', filename, reports])
proc.wait()
```

Currently it just zips up the directory `acceptance/reports` into the file `reports.zip`, but this still hides the layout from the build scripts. It takes the name of the ZIP file as its only argument, and it runs from the slave's `builder` directory. You should check this file in.

The build step is straightforward:

```
def reportsFile():
    return "reports.zip"

 def pythonAcceptanceBuilder(version):
    f.addStep(ShellCommand,
              haltOnFailure=True,
              command=[python_(version), "./tools/bin/accept.py"],
              description="Running acceptance tests",
              descriptionDone="Acceptance tests complete")
   f.addStep(ShellCommand,
              haltOnFailure=True,
              command=[python_(version),
                  "./tools/bin/package_reports.py",
                 reportsFile()],
              description="Packaging build reports",
              descriptionDone="Build reports packaged")
```

After you make this change, you should force a build, and it should succeed.

Retrieving the Reports

The Buildbot installation will have acceptance tests for both Python 2.4 and 2.5. These builds will run at the same time, so you need to keep the retrieved ZIP files separate. If you don't, then they may stomp on each other.

An easy way to do this is to create an upload directory for each builder. You'll name these directories `uploaded-py`*`Version`*. The version corresponds to the Python version (e.g., 2.5). These directories must be created before the copy happens.

The `FileUpload` build step copies the files. You supply the source file name on the slave, and the destination file name on the master. These can be absolute or relative path names. On the slave, they are relative to the build directory, and on the master, they are relative to the server root directory.

```
def reportsFile():
    return "reports.zip"

def makeUploadDirectory(version):
    uploads = uploadDirectory(version)
    if not os.path.exists(uploads):
       os.mkdir(uploads, 0750)

def reportsFileLocal(version):
    return os.path.join(uploadDirectory(version), reportsFile())

def uploadDirectory(version):
    return "uploaded-py%s" % version

def pythonAcceptanceBuilder(version):
    f.addStep(ShellCommand,
              haltOnFailure=True,
              command=[python_(version), "./tools/bin/accept.py"],
              description="Running acceptance tests",
              descriptionDone="Acceptance tests complete")
   f.addStep(ShellCommand,
              haltOnFailure=True,
              command=[python_(version),
                  "./tools/bin/package_reports.py",
                 reportsFile()],
              description="Packaging build reports",
              descriptionDone="Build reports packaged")
f.addStep(FileUpload,
              haltOnFailure=True,
              slavesrc=reportsFile(),
              masterdest=reportsFileLocal(version))
```

Once you've made these changes, you should restart the master and trigger a build. Once the build succeeds, you're ready to publish the reports.

Publishing the Reports

You'd think that unpacking a ZIP file to a directory would be an easy job. Unfortunately, you'd be wrong. The command you're trying to run is

```
unzip -qq -o -d %(destination)s %(zipfile)s
```

The -qq option silences all output, and the -o option overwrites existing files without prompting. The ZIP file will be unpacked in the directory specified by the -d option, which also happens to create any missing directories.

The destination location contains the revision number and the Python version. This path is public_html/rsreader/%(revision)s-py%(version), and it is relative to Buildbot's root directory on the master. Getting the revision number is the first hurdle.

Getting the Revision

The Buildbot 0.7.7 documentation suggests that this information is in the "revision" build property. BuildStep.getProperty() and BuildStep.setProperty() form the core of the build properties system, but only custom tasks can use them. ShellCommand classes have another access mechanism: command strings are wrapped in the WithProperties class, and this class expands them at runtime.[2]

The documentation suggests that the "revision" property is set by the SVN build step. Alas, that is not true. You must create a customized SVN build step to set this property.

SVN calls svn checkout to create the build directory. One of the last lines from this command is r'^Checked out revision \d+\.', where \d+ is the revision number. All you need to do is search the log for that pattern.

```
import re
from StringIO import StringIO
...
class SVNThatSetsRevisionProperty(SVN):

    checked_out_line = re.compile("^Checked out revision (\d+)\.")

    def createSummary(self, log):
        for line in StringIO(log.getText()).readlines():
            found = self.checked_out_line.search(line)
            if found:
                self.setProperty("revision", found.group(1))
        return SVN.createSummary(self, log)
```

The method createSummary(log) gives you access to the log just after it completes and just before Buildbot makes any decisions about the step's status. There are quite a few other hook methods that let you intercept a step's control flow.

It is time to replace the old SVN step with the new one:

```
def pythonBuilder(version):
    python = python_(version)
    nosetests = nosetests_(version)
    site_bin = site_bin_(version)
    site_pkgs = site_pkgs_(version)
```

2. Used thusly: ShellCommand(command=["rm", "-rf", **WithProperties**("uploaded-py%(revision)s")]).

```
    f = factory.BuildFactory()
    f.addStep(SVNThatSetsRevisionProperty,
       baseURL="svn://repos/rsreader/",
       defaultBranch="trunk",
       mode="clobber",
       timeout=3600)
    f.addStep(ShellCommand,
              command=["rm", "-rf", site_pkgs],
              description="removing old site-packages",
              descriptionDone="site-packages removed")
...
```

You'll see no change when you fire off the build this time.

Publishing the Build

The publishing step only runs on the master. Buildbot doesn't provide much support for this, but it's not too hard. You'll start with a very simple build step:

```
from buildbot.process.buildstep import BuildStep
from buildbot.status.builder import SUCCESS
...
class InstallReports(BuildStep):

    def start(self):
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

This step ties it into the build:

```
def pythonAcceptanceBuilder(version):
   ...
    f.addStep(FileUpload,
              haltOnFailure=True,
              slavesrc=reportsFile(),
              masterdest=reportsFileLocal(version))
    f.addStep(InstallReports, haltOnFailure=True)
```

Run it, and it should produce a successful green build step.

You'll need to publish to a URL. On my system, this URL is `http://buildmaster.theblobshop.com:8010/rsreader/%(revision)s-py%(version)s`. You'll use a fixed URL the first time, and you'll subsequently parameterize it:

```
class InstallReports(BuildStep):
    url = "http://buildmaster.theblobshop.com:8010/rsreader/18-py2.5"

    def start(self):
        self.setUrl("reports", self.url)
```

```
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

Parameterizing the URL requires the code revision and Python version. The customized SVN step supplies the revision via the build parameter. The build factory supplies the version as an argument, as with other steps in the build factory.

```
class InstallReports(BuildStep):
    url = "http://buildmaster.theblobshop.com:8010/rsreader/18-py2.5"

    def __init__(self, version, **kw):
        self.version = version
        BuildStep.__init__(self, **kw)
        self.addFactoryArguments(version=version)

    def start(self):
        self.setUrl("reports", self.url)
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

When you initialize a build step, you must always pass on the other arguments to the parent. Not doing this leads to unpredictable behavior.

Now you have to change how the factory calls the build step.

```
def pythonAcceptanceBuilder(version):
   ...
    f.addStep(FileUpload,
              haltOnFailure=True,
              slavesrc=reportsFile(),
              masterdest=reportsFileLocal(version))
    f.addStep(InstallReports,
              haltOnFailure=True,
              version=version)
```

Now the build step has access to the revision and version, so you can finally parameterize the URL:

```
class InstallReports(BuildStep):
    url = "http://buildmaster.theblobshop.com:8010" \
       "/rsreader/%(revision)s-py%(version)s"

    def __init__(self, version, **kw):
        self.version = version
        BuildStep.__init__(self, **kw)
        self.addFactoryArguments(version=version)
```

```
    def expansions(self):
        return {'revision': self.getProperty('revision'),
                'version': self.version}

    def start(self):
        self.setUrl("reports", self.url % self.expansions())
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

The step now expands the URL. You'll want the step to go yellow while it runs.

```
    def start(self):
        self.step_status.setColor("yellow")
        self.step_status.setText(["Publishing reports", "Unzipping package"])
        self.setUrl("reports", self.url % self.expansions())
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

Oh yeah, and you'll want to unzip the file too. There is a catch, though. Buildbot uses the Twisted framework. Twisted is an asynchronous interaction system, and it extensively manipulates operating system signals. This interferes with the normal subprocess calls, resulting in strange exceptions whenever you invoke any asynchronous operations—like checking a process's exit code.

Luckily, Twisted supplies process-handling methods. These methods include `getProcessOutput()`, `getProcessValue()`, and `getProcessOutputAndValue()`. These live in the package `twisted.internet.utils`. You'll use `getProcessValue()` first:

```
import os
from twisted.internet.utils import getProcessValue
...
class InstallReports(BuildStep):
    url = "http://buildmaster.theblobshop.com:8010/" \
       "rsreader/%(revision)s-py%(version)s"
    dest_path = "public_html/rsreader/%(revision)s-py%(version)s"
...
    def start(self):
        self.step_status.setColor("yellow")
        self.step_status.setText(["Publishing reports", "Unzipping package"])
        dest = self.dest_path % self.expansions()
        if not os.path.exists(dest):
            os.makedirs(dest, 0755)
        cmd = "/usr/bin/unzip"
        zipfile = os.path.abspath(reportsFileLocal(self.version))
        args = ("-qq",
                "-o",
                "-d", dest,
                zipfile)
```

```
        getProcessValue(cmd, args)
        self.setUrl("reports", self.url % self.expansions())
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)
```

When you run this step, it should succeed. If it succeeds, you'll find the results in the directory `public_html/rsreader/%(revision)s-py2.5/acceptance/reports`, under the Buildbot master's directory.[3] What if it doesn't succeed, though? You haven't checked the results of `getStatusValue()`, so you don't know. There's nothing special about checking the results, however:

```
from buildbot.status.builder import FAILURE, SUCCESS
...
    def start(self):
        self.step_status.setColor("yellow")
        self.step_status.setText(["Publishing reports", "Unzipping package"])
        dest = self.dest_path % self.expansions()
        if not os.path.exists(dest):
            os.makedirs(dest, 0755)
        cmd = "/usr/bin/unzip"
        zipfile = os.path.abspath(reportsFileLocal(self.version))
        args = ("-qq",
                  "-o",
                  "-d", dest,
                  zipfile)
        result = getProcessValue(cmd, args)
        if result == 0:
            self.setUrl("reports", self.url % self.expansions())
            self.step_status.setColor("green")
            self.step_status.setText("Reports published")
            self.finished(SUCCESS)
        else:
            self.step_status.setColor("red")
            self.step_status.setText("Report publication failed")
            self.finsished(FAILURE)
```

If your build failed before, then it's worth trying it again. You should see a red step without a URL this time. Once you have this step running, you can refactor it:

```
def start(self):
    self.begin()
    self.make_report_directory()
    results = self.unzip()
    if results == 0:
        self.succeed()
```

3. On my system, this is `/usr/local/buildbot/master/rsreader`.

```
        else:
            self.fail()

    def begin(self):
        self.step_status.setColor("yellow")
        self.step_status.setText(["Publishing reports", "Unzipping package"])

    def make_report_directory(self):
        dest = self.dest_path % self.expansions()
        if not os.path.exists(dest):
            os.makedirs(dest, 0755)

    def unzip(self):
        cmd = "/usr/bin/unzip"
        zipfile = os.path.abspath(reportsFileLocal(self.version))
        args = ("-qq",
                "-o",
                "-d", self.dest_path % self.expansions(),
                zipfile)
        return getProcessValue(cmd, args)

    def succeed(self):
        self.setUrl("reports", self.url % self.expansions())
        self.step_status.setColor("green")
        self.step_status.setText("Reports published")
        self.finished(SUCCESS)

    def fail(self):
        self.step_status.setColor("red")
        self.step_status.setText("Report publication failed")
        self.finsished(FAILURE)
```

As usual, you should verify the changes by running the build. You no longer need to run tests on this builder, so you can put it on a regular schedule.

Getting Regular Builds

Regularly timed builds are run with the `Nightly` scheduler. It runs builds at times specified by a combination of `dayOfMonth`, `dayOfWeek`, `month`, `hour`, and `minute`. If a value isn't specified, then it isn't used to match the date. The exception is `minute`, which defaults to `0`. If you've ever used the UNIX `cron` command, then you'll be right at home.

All of this makes more sense with a few examples. This will run every March at 6:42 PM and 9:42 PM:

```
month=3, hour=(18, 21), minute=42
```

This will run at 6:00 AM on every Monday that falls on the second or third day of the month:

```
dayOfMonth=(2, 3), dayOfWeek=0, hour=6
```

I like to run acceptance builds several times a day:

- Once in the morning so that people know the project status at the beginning of the day, including any changes that people made the night before
- Once at lunch to pick up the morning's work
- Once near the end of the day to pick up the afternoon's changes and report them before everyone goes home

Scheduling this takes just a few lines:

```
from buildbot.scheduler import Nightly, Scheduler
c['schedulers'] = []
c['schedulers'].append(Scheduler(name="rsreader under python 2.5",
                                 branch=None,
                                 treeStableTimer=5,
                                 builderNames=["buildbot-full-py2.5"]))
c['schedulers'].append(Scheduler(name="rsreader under python 2.4",
                             branch=None,
                             treeStableTimer=5,
                             builderNames=["buildbot-full-py2.4"]))
c['schedulers'].append(Nightly(name="python 2.5 acceptance builds",
                               builderNames=["buildbot-acceptance-py2.5"],
                               hour=(7, 12, 17), minute=0)
```

The odds are that you'll have to wait several hours for this to trigger a build. You can test `Nightly` by setting the hours and minutes to times just a minute or two in the future, but don't forget to restore them when you've finished your tests.

What's Left?

You went through a great deal of effort to ensure that a 2.4 builder could easily configured, but I'm not going walk you through the rest of the process. You should know more than enough at this point to set it up, and it's a great exercise. As you do, test each change. Buildbot configuration is a complicated path, and it's easy to get lost unless you keep track of each step.

Summary

FIT is a system for specifying and running functional tests. It consumes requirement documents with which it drives testing fixtures and produces reports. The system focuses upon the requirement documents. In an optimal situation, the customer produces them with the assistance of other team members. This process serves as the basis of detailed design discussions. Along the way, they create a common vocabulary, which can be considered the core of the system metaphor.

The requirement documents can also play the roles of design documents and testing plans. Developers and testers create test fixtures that connect these plans to the larger code base. Once the fixtures are developed, they can be run from the build. The output from these runs serves as a progress document for customers and for management. Watching as the requirements go from red to green over the course of an iteration instills them with confidence.

PyFit is the Python version of FIT. While the FIT framework theoretically copes with any kind of document that contains tables, PyFit only supports HTML. The documents are read from the filesystem or a FitNesse server. FitNesse is a wiki server for writing and running FIT tests. While appealing for very small projects, it doesn't cope well with branching, and disconnected operation isn't really possible. For these reasons, I prefer placing FIT tests into the source.

Tying PyFit into a build is theoretically an easy thing to do, but there are many intricacies. In the system you worked with, the tests run on the build slaves, and the results are packaged into ZIP files there, too. Scripts encapsulate the intricacies of both tasks and hide the details from the build server. The results are presented by the build master using Buildbot's internal web server. To do this, the ZIP file is copied to the build master, and then unpacked into the web server's document tree. This forms the basis for the project's dashboard.

Index

A

absolute paths, Buildbot, 124
accept.py build tool, 358
acceptance testing, 339, 340
  build to include acceptance tests, 353
  converting URLs into feed objects, 188
  creating directory for, 346
  description, 140, 173, 339
  Framework for Integrated Tests (FIT), 340–353, 367
    creating column fixture test with FIT, 346–353
  observing failing test first, 152
  PyFit, 340–353
    writing requirements, 341–344
  RSS reader application, 150, 152
  running FIT with Buildbot, 353–367
    defining Buildbot configuration file, 353
    getting regular builds, 366–367
    making reports available, 358–366
    preparing slave, 353
    running builder, 354–357
  running with every build, 339, 346
  using data files, 189
active record pattern
  ORMs, 266
  SQLObject, 267
add (__add__) method, 193
add command, Subversion, 50
add method, pMock, 201
add method, PyMock, 221
adding files, Subversion
  from command line, 50
  through Eclipse, 68
addRemoveName keyword
  multiple relationships, SQLObject, 282
addTestPage/addTestSuite methods
  aggregating JsUnit tests, 335–336
add_single_feed method, PyMock, 216–217
  refactoring, 218
admin_pw keyword, DBMigrate, 301
admin_user keyword, DBMigrate, 301
AggregateFeed class, pMock
  combine_feeds method, 199
  creating FeedEntry factory, 200
  empty AggregateFeed output, 209
  entries initialized to Set, 201
  feeds_from_urls method, 203
  initializing FeedParser factory, 203
  refactoring/extracting, 198
  reimplementing from_urls method, 204
AggregateFeed class, PyMock
  entries initialized to Set, 222
  print_entry_listings with empty feeds, 225
aggregating JsUnit tests, 335–336
aggregating two feeds, 194–195
aggregator, 149
agile development, 1, 3, 4
  databases and, 264, 296, 306
  impact on DBA organization, 264
  using IDEs with, 21
agile host, aliasing, 104
agile methods, 4
  collective code ownership, 12–13
  continuous integration, 16–17
  continuous reflection, 15
  documentation, 17
  on-site customers, 8–9
  pair programming, 5–7
  refactoring, 11–12
  short iterations, 13–15
  simple design, 12
  system metaphor, 8
  test-driven development, 10–11
  unit tests, 9
  user stories, 7–8
Ajax (Asynchronous JavaScript and XML), 312
aliasing hosts, 104
all (__all__) attribute, 182

all method, SQLAlchemy, 288
allow-hosts option, easy_install, 132
altinstall.pth file, 85–86
Ant Builder, 164
Apache plug-ins
  mod_python, 314
application class, 175
application frameworks
  connections to web servers, 313
application object, 315
application servers, Python, 313
application tests
  creating, 175
  implementing as native Nose tests, 176
applications
  full stack applications, 313
  web application frameworks, 313
  web applications, 312–320
arguments
  how dependencies arise, 190
  mocking calls to self, pMock, 196
Arguments field
  builder properties window, 165
  External Tools dialog, Eclipse, 169
argv file, sys
  running unit tests manually in Eclipse, 154
assert expression, Python
  success or failure of unit tests, 147
AssertionError, 147
assertStdoutEquals method
  running unit tests manually in Eclipse, 158
assert_equals method, 176
atoms
  creating migrations, DBMigrate, 302
attribute access, Python, 193
attribute defaults, SQLObject, 272
attribute setter mocking, PyMock, 229
attributes, HTML elements, 310
attributes, Nose, 160, 174
at_least playback count, PyMock, 229
at_least_once calling policy, pMock, 229
authority, organizational/technical, 245
autoflush attribute, SQLAlchemy, 286
autoformatting, 251
autoRun parameter, JsUnit
  running tests by URL, 337

B

backref keyword, SQLAlchemy, 294
baseURL property, Subversion, 119
bdist_egg command, Setuptools, 90
BeautifulSoup package, 320
Bicycle Repair Man program, 39
bin directory, Python
  Install step, Buildbot, 125
bin_dir function, PyFit, 352
Block Comments panel, Pydev, 252
branch coverage, 238
branches
  revision control systems, 44
browsers *see* web browsers
build automation
  continual verification of builds, 103
  replicable builds, 81
build command, Setuptools, 89
build directory
  building projects with setup.py, 88
  paths used on build slave, 124
  Subversion ignoring files, 98
build factories
  description, 109, 136
  encapsulating builder factory in function, 129
  running FIT with Buildbot, 353, 356
  supporting Python 2.4 builds, 129
  using source code, 119
build master, Buildbot, 107–112
  accessing Subversion repository, 116–118
  build slave contacting, 107
  configuring, 107
  description, 104, 136
  naming, 104
  options for triggering builds, 104
  running FIT with Buildbot, 354–355, 357
Build Options tab
  builder properties window, 165
build servers, Buildbot, 103–116
build slave, Buildbot, 112–116
  accessing Subversion repository, 116–118
  checking out code, 120
  configuring, 112
  contacting build master, 107
  description, 104, 136

installing Buildbot on buildmaster, 104
naming, 104
paths used on, 124
Python installation, 122
running FIT with Buildbot, 353
build steps, Buildbot
colors used for, 114, 116
Compile step, 124–125
description, 136
enhancing step progress description, 134
Install step, 125–128
running FIT with Buildbot, 353
SVN step, 119–121
build user, Buildbot
configuring build master, 107
configuring build slave, 112
Buildbot, 104–116
architecture, 104
build master, 104, 107–112
build slave, 104, 112–116
code coverage report, 259
Compile step, 124–125
configuring build system, 106
contrib directory, 121
coverage package, 258
enhancing step progress description, 134–136
hooks to Subversion events, 121
Install step, 125–128
installing, 104–106
installing contributed programs, 106
installing on Windows systems, 107
landing page, 111
notifications, 121
options for triggering builds, 104
paths used on build slave, 124
reconfig command, 119
reconfiguration, 120
running FIT with Buildbot, 353–367
defining Buildbot configuration file, 353
getting regular builds, 366–367
making reports available, 358–366
preparing slave, 353
running builder, 354–357
running full test suite at build time, 171–173
sendchange command, 114, 120
summary, 136
supporting Python 2.4 builds, 128–132
SVN step, 119–121
waterfall display, 111
buildbot binary
server startup script, 112
Buildbot.tac file
configuring build master, 108
configuring build slave, 113
buildbotUrl property, Buildbot, 110
builder
running FIT with Buildbot, 353, 354–357
builder directory
packaging reports, 359
paths used on build slave, 124
builder properties window, 164
tabs and fields, 165
builders, 109
preventing overlapping, 122
Builders menu item
project properties window, 163
builders property, Buildbot
configuring build master, 109
supporting Python 2.4 builds, 130
buildmaster
aliasing hosts, 104
configuring build slave, Buildbot, 112
configuring build system, 106
installing Buildbot on, 104
BuildmasterConfig dictionary, 108
builds
build to include acceptance tests, 353
building projects with setup.py, 88–91
continual verification of, 103
continuous builds, 340
formal builds, 340
incremental builds, Eclipse, 163
replicable builds, 85–86
path manipulation solution, 85
virtual Python solution, 85
running full test suite at build time, 171–173
supporting Python 2.4 builds, 128–132
when unit tests need to be run, 163
BuildSlave objects, Buildbot
configuring build master, 108
configuring build slave, 112

build_ext module, Setuptools, 89
build_py module, Setuptools, 89, 90

C

callbacks
  using write() callback, 315
camel case
  naming standards, 248
cascade keyword, SQLAlchemy, 295
Cascading Style Sheets (CSS), 310, 311
centralized revision control systems, 42
CGI (Common Gateway Interface), 313
chained expressions, SQLAlchemy, 290
checkout command, Subversion, 47
cheese shop, Python, 84
classes
  creating application class, 175
  creating Python class, 33
  subclassing, 193
code analysis features, Pydev, 253
code coverage, 237–239
  branch coverage, 238
  low test coverage, 238
  patching code coverage report into Buildbot, 259
  statement coverage, 237
Code Formatter panel, Pydev, 252
code reviews, 5, 249
code style options, Pydev, 252
coding
  collective code ownership, 12–13
  validating committed code, 256–257
coding conventions, 244
  consistency, 245
  features assisting writing/analyzing code, 251
  feedback on defective code, 251
  naming standards, 244
  Python, 246
coding standards
  autoformatting, 251
  code analysis features, 253
  rewarding good code, 248–249
  templates, Pydev, 254
cohesion, 141
  unit tests illustrating, 142
collective code ownership, 12–13
  unit tests, 143
column fixtures, 346
  representing business processes, 347
  tests created with FIT, 346–353
combine_feeds method, pMock
  AggregateFeed class, 199
  mocking calls to self, 196
  reimplementing from_urls method, 204
commit command, Subversion, 50
commit event, Subversion, 121
Commit window
  working with Subversion through Eclipse, 70
committers
  adding to Subversion group, 117
  build master/slave accessing Subversion, 116
  file ownership, 116
Common Gateway Interface (CGI), 313
Common tab
  External Tools dialog, Eclipse, 169
communication, developers, 250
Compile step, Buildbot, 124–125
complexity measurements, 239–242
components
  isolating components under test, 190–192
configure step
  supporting Python 2.4 builds, 128
connections
  application frameworks to web servers, 313
connect_application method
  running DBMigrate with unit tests, 305
consistency
  coding conventions, 245
Console Encoding field
  Common tab, External Tools dialog, 170
Console view
  working with Subversion through Eclipse, 65
console_scripts key, setup.py, 91
continuous builds, 340
continuous integration, 16–17
  replicable builds, 85
continuous reflection, 15
contrib directory, Buildbot, 121
copy command, Subversion, 51
  working with Subversion through Eclipse, 78

copying files, Subversion
working from command line, 51
working through Eclipse, 78
coupling, 141, 142
coverage package, 258
coverage report weakness, 259
running coverage through Nose, 258
with-coverage option, nosetests, 258
createSummary method
publishing reports, 361
create_entry method, pMock, 200
create_entry method, PyMock, 217, 218, 220, 221
create_schema method
schema definition, SQLObject, 272
crontab command, 112
CSS (Cascading Style Sheets), 310, 311
customer tests, 139, 173
cyclomatic complexity, 239, 241

D

d option, unzip command
publishing reports, 360
daemon option, svnserve, 117
data mapper pattern, ORMs, 266
database administrators (DBAs), 263, 265
database migrations
testing databases, 298
databases
adjusting between database environments, 296
agile development, 264, 296, 306
background, 306
background to working with, 263
connecting to databases
concealing data access, 265
DBMigrate, 301
SQLAlchemy, 283
SQLObject, 268
creating rows, SQLObject, 269–272
evolution of database design, 264
isolation, 264
migrations, 298–306
DBMigrate, 300–306
generating migration instructions, 299
locating migration mechanism, 300
numbering migrations and playback, 299
object-relational mappers (ORMs), 265–267
Python, 267–296
SQLAlchemy, 283–296
SQLObject, 267–282
refactoring, 264, 298
testing, 264
db keyword, DBMigrate, 301
DBAs (database administrators)
working with databases, 263
working with developers, 265
DBMigrate, 300–306
admin_pw keyword, 301
admin_user keyword, 301
atoms, 302
connecting to database, 301
creating migrations, 301–303
db keyword, 301
functions, 302
host keyword, 301
installing, 300
manually migrating database, 303–304
migration dictionary, 303
port keyword, 301
pw keyword, 301
revision flag, 304
running DBMigrate from program, 305
running DBMigrate with unit tests, 305
scheme keyword, 301
socket keyword, 301
string expansion, 302
user keyword, 301
verbose flag, 304
versiontable keyword, 301
debugging
unit testing to reduce, 141
value of debuggers, 141
decorators
creating rows, SQLObject, 271
decorators, Python, 161
use_pymock decorator, PyMock, 212
default attribute, SQLObject, 273
defaultBranch property, Subversion, 119
delete command, Subversion, 52
working with Subversion through Eclipse, 77
delete method, SQLAlchemy, 295

deleting files, Subversion
  working from command line, 52
  working through Eclipse, 76
deleting rows, SQLObject, 275
dependencies
  application initializing dependencies, pMock, 211
  ensuring local dependency processing, 132
  how dependencies arise, 190
  isolating components for testing, 191
  Python packages, 83
  Setuptools managing dependencies, 92–94
  testing and, 142
  test_rsreader_dependency_initialization test, PyMock, 227
  verifying dependencies initialized correctly, pMock, 206
description/descriptionDone keywords, Buildbot, 134
design
  iterative design methods, 175
  simple design, 12
destroySelf method
  deleting rows, SQLObject, 275
developers
  communication, 250
  kinds of feedback for development, 234
  pair programming, 249
  rewarding good code, 248–249
  why developers need feedback, 234
  working with databases, 263
  working with DBAs, 265
development
  see also agile development
  code reviews, 249
  goal of development organization, 263
  using production database, 297
development environments, 22
development methodologies
  agile development, 1
  iterative methodologies, 3
  reasons for, 1
  test-driven development (TDD), 10–11, 146–147
  waterfall methodology, 2
development mode, Setuptools, 100–102
development processes
  benefits of version/revision control, 41
  collective code ownership, 12–13
  continuous integration, 16–17
  continuous reflection, 15
  documentation, 17
  on-site customers, 8–9
  pair programming, 5–7
  refactoring, 11–12
  short iterations, 13
  system metaphor, 8
  user stories, 7–8
development velocity, 242–243
diff command, Subversion, 54–56
directories, Eclipse, 38
directory structure, Subversion, 44
disconnect_application method
  running DBMigrate with unit tests, 305
Display in favorites menu field
  Common tab, External Tools dialog, 170
dist directory
  Setuptools, 91
  Subversion ignoring files, 98
distributed mode, JsUnit, 337
distributed revision control systems, 42
Distutils library, 81
  description, 102
  installing Python packages, 83
distutils.cfg file, 86
Docstrings panel, Pydev, 252
Document Object Model (DOM), 328
documentation
  agile methods, 17
  developer communication, 250
docutils package
  Install step, Buildbot, 126
DOM (Document Object Model), 328
DSL (domain-specific languages), 193
duck typing, 193
dummies, 191
  writing in Python, 192
dummy objects, PyMock, 216

E

EasyMock
  record-replay, 193
easy_install program, Setuptools, 87
  allow-hosts option, 132
  development mode, 100
  installing Buildbot, 104, 106
  installing from local copy, 96
  removing existing package, 95
echo step output
  configuring build slave, Buildbot, 115
Eclipse
  advantages of, 21
  creating Python development environment, 40
  development environments, 22
  directories, 38
  ignoring files, 100
  importing from Subversion, 60–64
  incremental builds, 163
  installing, 25
  installing Mylyn, 25–31
  installing plug-ins, 30
  installing Pydev, 31–32
  job management system, 21
  perspectives, 24
  project file, 38
  pydevproject file, 38
  revision control plug-ins, 59
  revision control systems and, 59–64
  running full test suite in development, 168
  running unit tests manually in, 151–159
  selecting workspace root, 23
  shared projects, 59
  sharing subverted project, 59–60
  source folders, 33
  src directory, 39
  startup screen, 24
  views, 24
  workbench, 25
  working with Python, 25
  working with Subversion through, 64–79
Eclipse plug-ins
  Pydev, 22, 25
  Pydev Extensions, 22
  SQLExplorer, 22
  Subversive, 22
Eclipse Preferences window, 251
edit-and-merge process
  revision control systems, 43
  Subversion and, 79
editing files
  working with Subversion through Eclipse, 71
eggs
  installing, 84
  installing Setuptools, 86
  naming structure, 84
  setup.py creating, 87
  Setuptools and, 81
egg_info command, Setuptools, 90
elements, HTML, 310
ElementTree package
  MVC testing markup, 318–319
  XPath operations supported by, 319
embedded code
  testing databases, 298
encapsulation, 141
encryption
  https scheme, 311
ending keyword, PyMock, 230
entities, global
  how dependencies arise, 191
entry_listings method, pMock, 207–208
entry_listings method, PyMock, 223
entry_points attribute, setup.py, 91
environ dictionary
  WSGI conversations, 315
eq constraint, PyMock, 213
errors
  branch coverage, 238
  JsUnit, 331
estimates
  short iterations, 14
exceptions, pMock/PyMock, 228
exclusive locking
  revision control systems, 43
executables
  Install step, Buildbot, 125
  paths used on build slave, 124
expectations, PyMock
  alternative syntax to define, 214
expected_items value, 182
expected_line value, 176

expects clause
  defining expectations, PyMock, 215
  pMock, 196
    playback counts, 229
exploratory testing, 140
extent, unit tests, 145
External Tools dialog, Eclipse, 168–169
external_requirements attribute, setup.py, 92
ez_setup.py program
  bootstrapping Setuptools, 97
  installing Setuptools, 86
  using local copy of Setuptools, 132

F

factories *see* build factories
failures, assertions
  JsUnit, 331
fakes, 192
Feature License screen
  installing Mylyn, 27
Feature Verification screen
  installing Mylyn, 30
feedback
  defective code, 251
  environmental feedback, 234
  kinds of feedback for development, 234
  measurements, 235–236, 261
  social feedback, 234
  why developers need feedback, 234
FeedEntry constructor, PyMock
  mocking __init__ directly, 222
  test_feed_entry_constructor test, 223
  test_feed_entry_listing test, 222
FeedEntry factory, pMock
  AggregateFeed creating, 200
  from_parsed_feed method, 202
  listing method, 203
  test_feed_entry_listing test, 202
  verifying operation, 202
FeedParser factory, pMock
  AggregateFeed initializing, 203
FeedParser package, 149
  converting URLs into feed objects, 185
feeds
  add_single_feed method, PyMock, 216, 217
  aggregating two feeds, 194–195
  converting URLs into feed objects, 185, 188
  creating aggregate entries for feeds, pMock, 197
  formatting feed entry listings, pMock, 207
  from_urls method, pMock, 204
  taking and converting, pMock, 200
feeds_from_urls method, pMock, 204
  AggregateFeed initializing FeedParser factory, 203
  FeedEntry factory, 203
  reimplementing from_urls method, 204
FeedWriter class, pMock
  formatting feed entry listings, 207
  initializing stdout attribute, 209
feed_from_url method
  converting URLs into feed objects, 185
  isolating components, 190
feed_listing method, 180
fields
  updating fields, SQLObject, 274
file repository *see* repository
file revisions, storage of, 40
file scheme, 311
File Types panel, Pydev, 252
files
  job management system, 21
FileUpload build step
  retrieving reports, 360
filter method, SQLAlchemy, 289
filter_by method, SQLAlchemy, 290
find-links option, Setuptools
  fixing options with setup.cfg, 97
findall method
  MVC testing markup, 319, 320
findtext method
  MVC testing markup, 319
find_dev_root function
  running PyFit, 352
find_packages function, Setuptools, 88
first method, SQLAlchemy, 288
FIT (Framework for Integrated Tests), 340–367
  background, 340
  components, 340
  creating column fixture test with, 346–353
  description, 367

families of tests created with, 346
FIT report for broken test, 344
FitNesse, 340, 368
fixtures, 341
PyFit and, 368
PyFit example, 344–345
reports, 341
requirement documents, 340, 341, 346
writing requirements, 341–344
running FIT with Buildbot, 353–367
defining Buildbot configuration file, 353
getting regular builds, 366–367
making reports available, 358–366
preparing slave, 353
running builder, 354–357
showing successful/unsuccessful test results, 344
test runners, 341
XML summary document produced, 343
FitNesse, 340, 368
fixtures
column fixtures, 346
creating column fixture test with FIT, 346–353
description, 348
Framework for Integrated Tests (FIT), 341
writing HTML spec document, 343
parts of, 348
flushing
autoflush attribute, SQLAlchemy, 286
FolderRunner.py
creating column fixture test with FIT, 349
running PyFit, 351
foreign keys, SQLAlchemy, 291, 292
foreign keys, SQLObject, 275
formal builds, 340
Framework for Integrated Tests *see* FIT
frameworks
web application frameworks, 337
from_parsed_feed method, pMock
FeedEntry factory, 202
formatting feed entry listings, 208
from_statement method, SQLAlchemy, 290
from_urls method, pMock, 204
from_urls method, PyMock, 215–216, 219
full stack applications, 313
functional testing
databases, 298
description, 139, 339
Framework for Integrated Tests (FIT), 340–353, 367
running FIT with Buildbot, 353–367
running tests with every build, 340
size of functional test suite, 353
functions
creating migrations, DBMigrate, 302
decorators, 161
how dependencies arise, 190

G

generator mocking, PyMock, 230
get method
retrieving objects, SQLObject, 273
getElementById method, JavaScript, 335
getitem (__getitem__) function, 193
PyMock, 213
getProcessValue method, Twisted, 364
getProperty method, BuildStep class, 361
global entities
how dependencies arise, 191
graphics
MVC testing web applications, 317
groups
adding committers to Subversion group, 117
gui_scripts key, setup.py, 91

H

harness *see* test harness
hooks
hooks sending notifications, Buildbot, 121
main method, 175
precommit hooks, 256
host keyword, DBMigrate, 301
hosts
aliasing hosts, 104
allow-hosts option, easy_install, 132
HTML (Hypertext Markup Language), 310–311
HTML documents and forms, 337
SGML and, 310
Web and, 309, 337
writing HTML spec document, 342

html.WebStatus class
 configuring build master, Buildbot, 110
HTMLParser library
 MVC testing markup, 317
HTTP (Hypertext Transfer Protocol), 312
 state, 312
 Web and, 309, 337
http scheme, 311
https scheme, 311
http_port keyword
 configuring build master, Buildbot, 110

I

IDEs
 development environments, 22
ignore property, Subversion, 99
ignoring files, Eclipse, 100
ignoring files, Subversion, 98–100
images
 MVC testing web applications, 317
impersonators *see* impostors
import command, Subversion, 46
Import project window
 importing from Subversion, 60
imports
 monkeypatching and, 186
impostors
 categories of, 191
 description, 231
 dummies, 191
 fakes, 192
 mock objects, 192
 mocking to break dependencies, 191
 stubs, 192
 writing impostors, 192
incremental builds, Eclipse, 163
indentation settings
 autoformatting, Pydev, 252
info command, Subversion, 49
 verifying status, 117
info directory
 configuring build slave, Buildbot, 113
init (__init__) method
 application initializing dependencies, pMock, 211
 creating FeedEntry factory, pMock, 201
 mocking __init__ directly, PyMock, 222
install command, Setuptools, 90
 ensuring local dependency processing, 132
 installing from local copy, 96
Install step, Buildbot, 125–128
 ensuring local dependency processing, 134
installations
 Buildbot, 104–106
 DBMigrate, 300
 Eclipse, 25
 Eclipse plug-ins, 30
 eggs, 84
 JsUnit, 321
 Mylyn, 25
 pMock, 195
 Pydev, 32
 Setuptools, 86–87
 SQLAlchemy, 283
 SQLObject, 267
 Subversion, 44–47
 Twisted, 104
InstallReports class, 362
install_lib command, Setuptools, 90
install_requires attribute, setup.py, 92
integration testing
 description, 140, 339
 Framework for Integrated Tests (FIT), 340–353
 running FIT with Buildbot, 353–367
intermediateTable keyword, SQLObject, 282
interpreter, Python
 paths used on build slave, 124
isolating components under test, 190–192
isolation, databases, 264
is_empty method, pMock, 210
is_empty method, PyMock, 225, 226
iterative design methods, 175
iterative methodologies, 3

J

JavaScript, 312
 getElementById method, 335
 HTML documents and forms, 337
 interacting with browser, 328
 NaN value, 327
 null value, 327

pressCalculate method, 335
setUp function, 330
"Show all" errors window, 332
suite function, 335
testing, 320, 326
undefined value, 327
unit testing, 337
validateSlope method, 331, 335
validation function, 329
variable declaration/assignment, 327
jMock, 193
job management system, 21
join method, SQLAlchemy, 295
join statements, SQLObject, 279
joinColumn keyword, SQLObject, 282
joins, SQLObject, 275, 276
JsUnit, 321–326
adjusting timeouts, 324
aggregating tests, 335–336
author, 321
correspondence with unittest, 326
description, 337
distributed mode, 337
failures and errors, 331
installing, 321
package, 321
running tests by URL, 336
stand-alone testing, 322
test runner, 322
Jython, 33

K

keys
foreign keys, SQLObject, 275
keyword queries, SQLAlchemy, 290
Komodo IDE, 22

L

labels
revision control systems, 44
landing page, Buildbot, 111
libraries
mocking libraries, 193–194
library directories, Pydev, 33
license agreements
installing Eclipse plug-ins, 28
linear independence
cyclomatic complexity, 239
Linux
usermod command, 117
list command, Subversion, 46
listing method, pMock
FeedEntry factory, 203
listing_from_item method, 176
list_from_item method, 177
load testing, 140
local dependencies
ensuring local dependency processing, 132
Location field
builder properties window, 165
External Tools dialog, Eclipse, 169
log command, Subversion, 56
low test coverage, 238

M

MailNotifier class
configuring build master, Buildbot, 110
mailto scheme, 311
main method
converting URLs into feed objects, 188
hook between Setuptools and application class, 175
implementing application class, 176
new_main method, pMock, 210, 212
test_main test, pMock, 212
test_main test, PyMock, 227
Main tab, builder properties window, 165
makefiles, Buildbot
configuring build master, 108
configuring build slave, 113
many-to-many relationships
SQLAlchemy, 293
SQLObject, 277
mapper directive
one-to-many relationships, SQLAlchemy, 292
mapping layer database interactions, 297
markup
MVC testing web applications, 317–320
master *see* build master, Buildbot
master.cfg file
configuring build master, Buildbot, 108
paths used on build slave, 124
running FIT with Buildbot, 353, 354
using source code, 119

McCabe complexity *see* cyclomatic complexity
measurements, 236–237
  code coverage, 237–239
  coding conventions, 244
  complexity measurements, 239–242
  development velocity, 242–243
  effecting change, 236
  factors characterizing, 236
  feedback, 235, 236, 261
  instrument for measuring, 236
  low test coverage, 238
  purpose of, 236
  qualitative measurements, 235, 243–246, 261
  quantitative measurements, 235, 237–243, 261
  scope of, 236
  side effects, 236–237
  software quality, 235
  units of measurement, 236
  what is being measured, 236
merge command, Subversion, 56
merging files
  revision control systems, 43–44
MetaData class, SQLAlchemy, 284
method function, PyMock, 215
methodologies *see* development methodologies
methods
  agile methods, 4
  decorators modifying, 161
  how dependencies arise, 190
  overriding existing methods, 185–189
  protocols, 193
  testing and, 142
middleware, WSGI, 316
migration dictionary, DBMigrate, 303
migrations, database, 298–306
  DBMigrate, 300–306
  generating migration instructions, 299
  locating migration mechanism, 300
  numbering migrations and playback, 299
  testing databases, 298
mkdir command, Subversion, 46
mock object packages
  types (pMock/PyMock) compared, 231
mock objects
  attribute setter mocking, PyMock, 229
  benefit of, 206
  code coverage, 238
  description, 192
  domain-specific languages (DSL), 193
  EasyMock, Java, 193
  exception mocking, pMock, 228
  exception mocking, PyMock, 228
  generator mocking, PyMock, 230
  introducing into tested code, pMock, 196
  mocking class constructors, PyMock, 220
  pMock, 194, 195–212
  PyMock, 212–228
  specifying calls on mock objects, PyMock, 215
  switching from record mode to replay mode, PyMock, 213
  verify method, PyMock, 213
mocking
  breaking dependencies, 191
mocking libraries, 193–194
model-view-controller (MVC), 316, 337
modules
  mocking external modules, PyMock, 216
modules, Pydev, 33
modules, Python, 82
  coupling, 141
mod_python, Apache plug-in, 314
monkeypatching, 185–189
  description, 231
  drawbacks if hand coding, 192
  imports and, 186
  mocking calls to self, pMock, 196
  pMock, 193
  PyMock, 214–215
    moving methods to new object, 218
    objects that Python can't monkeypatch, 224
  writing impostors, 192
motivation
  rewarding good code, 248–249
move command, Subversion, 51
moving files, Subversion
  working from command line, 51
  working through Eclipse, 77
multiple joins, SQLObject, 276

multiple relationships, SQLObject, 280
MVC (model-view-controller)
testing web applications, 316, 337
Mylyn
advantages of Eclipse IDE, 21
installing, 25
job management system, 22

N

name attribute, setup.py, 87
naming conventions/standards
camel case, 248
coding conventions, 244
renaming, 183–184, 250
typographical naming conventions, 248
NaN value, JavaScript, 327
native tests, Nose, 149
never calling policy, pMock, 229
new_main method, pMock, 210, 212
new_main method, PyMock, 226
Nightly scheduler, 366
Node Kind field
info command, Subversion, 49
nodes, HTML, 310
Nose
attributes, 160, 174
coverage package, 258
description, 149, 174
discovering unit tests, 149
finding tests with Nose, 159–160
implementing application tests as Nose tests, 176
introduction, 148
native tests, 149
options in builder properties window, 165
running coverage through Nose, 258
running full test suite in development, 167
skipping slow tests, 160–162
stdout and stderr, 160
tags, 160
tests_require property, 167
tests_suite property, 167
turning off output capture, 160
underscore (_) prefix, 195
nosetests, 159
builder properties window, 165
negating options, 161
standard test output, 160
vociferous test output, 160
with-coverage option, 258
notification classes
configuring build master, Buildbot, 110
notifications, 121
null value, JavaScript, 327
nullable keyword, SQLAlchemy, 285

O

-o option, unzip command, 360–361
object-relational mappers *see* ORMs
object-relational mismatch, 264
on-site customers, 8–9
once calling policy, pMock, 229
once playback count, PyMock, 229
one method, SQLAlchemy, 288
one-to-many relationships
SQLAlchemy, 291
SQLObject, 275
one_or_more playback count, PyMock, 229
operator overloading, Python, 193
organizational authority, 245
ORMs (object-relational mappers), 265–267
active record pattern, 266
aspects of ORMs, 265
data mapper pattern, 266
description, 306
Python ORMs, 267–296
SQLAlchemy, 283–296
SQLObject, 267–282
unit of work pattern, 266
otherColumn keyword, SQLObject, 282
overloading operators, Python, 193
override expression, PyMock, 215
override function, PyMock, 215–216
monkeypatching, 214
overriding objects
monkeypatching and imports, 186

P

packages
catching missing packages, 132
coverage package, 258
FeedParser package, 149
Install step, Buildbot, 125
paths used on build slave, 124
packages directive, Setuptools, 88
packages, Pydev, 33

packages, Python, 82
egg naming structure, 84
installing, 83–84
managing dependencies, 83
packages for unit testing, 148
packages, Setuptools
finding packages on the Net, 94
managing dependencies, 92–94
removing existing package, 95
package_dir directive, Setuptools, 88
package_reports.py tool, 358
pair programming, 5–7, 249
params dictionary
running DBMigrate from program, 306
parsing
FeedParser package, 149
HTMLParser library, 318
MVC testing markup, 318
path manipulation solution
replicable builds, 85
paths, Buildbot
absolute paths, 124
paths used on build slave, 124
relative paths, 124
Pattern field, Pydev templates, 255
PBChangeSource class
configuring build master, Buildbot, 109
pending prefix
AggregateFeed creating FeedEntry factory, pMock, 200
testing, 195
PEPs (Python Enhancement Proposals), 247
PEP 20 - The Zen of Python, 247
PEP 257 - Docstring Conventions, 248
PEP 333 - Web Server Gateway Interface, 314
PEP 8 - Style Guide for Python Code, 248
performance testing, 140, 339
perspectives, Eclipse, 24
phytoplankton host, aliasing, 104
planning
agile development and, 4
playback counts, pMock/PyMock, 229
plug-ins
installing Eclipse plug-ins, 30
pMock, 195–212
AggregateFeed creating FeedEntry factory, 200
AggregateFeed initializing FeedParser factory, 203
AggregateFeed.entries initialized to Set, 201
application initializing dependencies, 211
creating aggregate entries for feeds, 197
description, 193–194
empty AggregateFeed output, 209
exception mocking, 228
expects clause, 196
FeedWriter initializing stdout attribute, 209
formatting feed entry listings, 207
installing, 195
introducing mocks into tested code, 196
mock object packages compared, 231
mocking calls to self, 196
modeling basis for, 175, 231
playback counts, 229
raising exceptions with pMock, 228
refactoring/condensing tests, 206
refactoring/extracting AggregateFeed, 198
same constraint, 196
taking and converting feeds, 200
test_add, 201
test_add_single_feed, 197, 199
test_aggregate_feed_creates_factory, 206
test_aggregate_feed_initializes_feed_parser test, 203, 206
test_combine_feeds, 196
test_create_entry, 200
test_entries_is_always_defined, 201
test_feed_entry_from_parsed_feed, 208
test_feed_entry_listing, 202
test_feed_writer_intializes_stdout, 209
test_feed_writer_prints_nothing_with_an_empty_feed, 209
test_from_urls, 204
test_get_feeds_from_urls, 203
test_is_empty, 210
test_main, 212
test_new_main, 210
test_print_entry_listings, 208, 210
test_rsreader_initializes_dependencies, 211
turning new_main into main method, 212
verifying dependencies initialized correctly, 206

verifying FeedEntry factory operation, 202
will clause, 196
pMock methods
add method, 201
combine_feeds method, 199
create_entry method, 200
defining feeds_from_urls method, 203
feeds_from_urls method, 204
feed_entry_listing method, 202
from_parsed_feed method, 202, 208
from_urls method, 204
is_empty method, 210
new_main method, 210, 212
print_entry_listings method, 208, 210
verify method, 196
port keyword, DBMigrate, 301
post-commit hook, Buildbot, 121
Postel's Law, 311
pre-commit hook, Buildbot, 121
pre-commit hooks, 256–257
pressCalculate method, JavaScript, 335
primary keys, SQLAlchemy, 284
printed_items value, 179
breaking into strings, 182
renaming, 183
print_entry_listings method
pMock, 208, 210
PyMock, 224, 225
processConnection variable
setting up SQLite connection, 269
production database
using for development, 297
programmer tests, 139
see also unit tests
programming
cohesion, 141
collective code ownership, 12–13
coupling, 141
pair programming, 249
web application frameworks, 337
project file, Eclipse, 38
project major/minor, 45
project properties window, 163
projectName property, Buildbot, 110, 112
projects
building projects with setup.py, 88–91
Import project window, 60
job management system, 21
organizing projects, 45
shared projects, 59
starting new Pydev project, 32–38
projectUrl property, Buildbot, 110, 112
properties, Subversion
ignoring files, 99
propset command, Subversion, 99
protocols, Python, 193
protocols, Web, 312
public_html directory, Buildbot, 108
pw keyword, DBMigrate, 301
PyChecker, 256
Pydev
autoformatting, 251
creating Python class, 33
Eclipse, 25
Eclipse plug-ins, 22
features assisting writing/analyzing code, 251
installing, 32
library directories, 33
modules, 33
packages, 33
renaming unit test values for readability, 183
starting new project, 32–38
templates, 254
Pydev extensions, 39
code analysis features, 253
Eclipse plug-ins, 22
Pydev Package Explorer pane, 33
working with Subversion through Eclipse, 68
pydevproject file, Eclipse, 38
pydistutils.cfg file, 85, 86
PyFit, 340–353
creating column fixture test with FIT, 346–353
running PyFit, 349–353
creating directory for acceptance tests, 346
description, 368
example, 344–345
FitNesse, 368
fixtures, 348
programs for running tests, 349

requirement documents, 341–344
running FIT with Buildbot, 357
making reports available, 358
spreadsheet support, 341
PyLint, 256
PyMock, 212–228
activating new functionality, 227
AggregateFeed.entries initialized to Set, 222
AggregateFeed.__init__, 221
alternative syntax to define expectations, 214
attribute setter mocking, 229
dummy objects, 216
ending keyword, 230
entry listings sorting test, 223
eq constraint, 213
exception mocking, 228
expects clause, 215
FeedEntry.__init__, 222
generator mocking, 230
getitem (__getitem__) function, 213
method function, 215
mock object packages compared, 231
mocking class constructors, 220
mocking external modules, 216
mocking __init__ directly, 222
modeling basis for, 175, 231
monkeypatching, 214–215
moving methods to new object, 218
objects that Python can't monkeypatch, 224
override function, 215–216
playback counts, 229
raising exceptions with PyMock, 228
record-replay model, 213
replay mode, 213
returns clause, 215
RSReader initialization test, 227
specifying calls on mock objects, 215
switching from record mode to replay mode, 213
test_add, 221
test_add_single_feed, 217
test_entries_is_always_defined, 221
test_feed_entry_constructor, 222–223
test_feed_entry_listing, 222
test_from_urls, 216, 219
test_getting_attributes, 229
test_is_empty, 226
test_main, 227
test_new_main, 226
test_print_agg_feed_listing_is_printed, 224
test_rsreader_dependency_initialization, 227
test_setting_attributes, 229
use_pymock decorator, 212
using PyMock with unittest, 230
PyMock methods
add method, 221
add_single_feed method, 216–217, 218
create_entry method, 217–218, 220–221
entry_listings method, 223
from_urls method, 215–216, 219
print_entry_listings method, 224, 225
verify method, 213
PyTest, 149
Python
build slaves and, 122
cheese shop repository, 84
coding standards, 246
creating classes, 33
decorators, 161
egg naming structure, 84
ez_setup.py program, 86
global entities, 191
modules, 82
ORMs, 267–296
package root, 88
packages, 82
installing, 83–84
managing dependencies, 83
unit testing, 148
protocols, 193
site package mechanism, 86
types of defective coding, 251
typographical naming conventions, 248
Python 2.4
supporting Python 2.4 builds, 128–132
Python application servers, 313
Python development environment
creating for Eclipse IDE, 40
Python Enhancement Proposals *see* PEPs

Python interpreter, 124
Python Interpreters screen, 32
PYTHONPATH variable, 32
  installing eggs, 84
  replicable builds, 85
  running PyFit, 350–351
  specifying additional packages, 82

Q

-qq option, unzip command, 360
quality, software, 234
qualitative measurements, 243–246
  coding conventions, 244
  description, 235, 261
quantitative measurements, 237–243
  code coverage, 237–239
  complexity measurements, 239–242
  description, 235, 261
  development velocity, 242–243
queries, SQLAlchemy, 287
  chained expressions, 290
  filtering with SQL queries, 289
  keyword queries, 290
  querying relations, 295
query method, SQLAlchemy, 287
  chained expressions, 290

R

raises method, PyMock, 228
raise_exception function, pMock/PyMock, 228
readability
  renaming, 183
reconfig command, Buildbot, 119
reconfiguration, Buildbot, 120
record mode, PyMock, 213
record-replay, EasyMock, 193
record-replay model, pMock, 194
record-replay model, PyMock, 213
refactoring, 11–12
  add_single_feed method, PyMock, 218
  condensing tests, pMock, 206
  creating rows, SQLObject, 271
  databases, 264, 298
  extracting AggregateFeed, pMock, 198
  implementing application class, 176
  moving from_urls, PyMock, 219
  moving methods to new object, PyMock, 218
  reasons for, 143
  reimplementing from_urls, pMock, 204
  running unit tests manually in Eclipse, 155–156
  triangulation, 182
  turning new_main into main method, pMock, 212
refactoring menu, Pydev renaming capability, 183
refactoring preview window, Pydev, 184
refactoring tools, 250
references to objects
  monkeypatching and imports, 186–187
regression testing, 10, 140
relationships
  many-to-many, SQLAlchemy, 293
  many-to-many, SQLObject, 277
  multiple, SQLObject, 280
  one-to-many, SQLAlchemy, 291
  one-to-many, SQLObject, 275
  querying, SQLAlchemy, 295
relative paths, Buildbot, 124
relocate option, switch command, 118
relying on the compiler technique, 178
Rename refactoring window, Pydev, 184
renaming, 183–184, 250
  working with Subversion through Eclipse, 77
replay mode, PyMock, 213
replicable builds, 85–86
reports
  Framework for Integrated Tests (FIT), 341
  running FIT with Buildbot, 358–366
    getting revision, 361–362
    making reports available, 358
    packaging reports, 358–359
    publishing build, 362–366
    publishing reports, 360–366
    retrieving reports, 359–360
repository
  continuous integration, 16
repository, Python
  cheese shop, 84

repository, Subversion
accessing, 45
creating, 44
defining repository location, 61
repository tree, 116
team repository view, 65–68
requests, HTTP, 312
requirement documents
design and testing, 368
Framework for Integrated Tests (FIT), 340–341, 346
writing requirements, 341–344
writing HTML spec document, 342
resolved command, Subversion, 58
responses, HTTP, 312
results, SQLAlchemy, 287
return values
how dependencies arise, 191
returns clause, PyMock
defining expectations, 215
revert command, Subversion, 53
Revert window
working with Subversion through Eclipse, 73
reviews
code reviews, 249
short iterations, 13
revision control systems
benefits of, 41
benefits of Subversion, 42–44
branches, 44
distributed revision control systems, 42
edit-and-merge process, 43
exclusive locking, 43
labels, 44
merging files, 43, 44
subverting Eclipse, 59–64
types of, 42
working with Subversion from command line, 47–59
working with Subversion through Eclipse, 64–79
Revision field
info command, Subversion, 49
revision flag, DBMigrate, 304
revision property
publishing reports, 361
revisions keyword
running DBMigrate from program, 306
RFC 2616, 312
RFC 3875, 313
RFC 822, 312
rows, SQLObject
creating, 269–272
deleting, 275
RSReader application
acceptance tests, 152
application initializing dependencies, pMock, 211
building project, 88–91
building RSS reader, 81
converting URLs into feed objects, 185, 188
finding packages, 88
identifying project with name attribute, 87
implementing application class, 176
initialization, PyMock, 227
installing executables using setup.py, 91
installing rsreader into site-packages, 89
managing dependencies, 92–94
test_rsreader_dependency_initialization, PyMock, 227
test_rsreader_initializes_dependencies, pMock, 211
using data files, 189
with unit test skeleton, 152
rsreader directory
paths used on build slave, 124
RSReader links, Buildbot, 112
rsreader-linux slave, Buildbot, 108
RSS feeds, 149
RSS reader application
acceptance testing, 150
aggregators and, 149
building simple RSS reader, 81
running commands, Setuptools, 89

S

same constraint, pMock, 196
Save as field
Common tab, External Tools dialog, 170
scheduler
running FIT with Buildbot, 353, 354

schedulers property, Buildbot
  configuring build master, 109
  supporting Python 2.4 builds, 130
schema definition, SQLAlchemy, 284
schema definition, SQLObject, 272
scheme keyword, DBMigrate, 301
scrum methodology, 243
secondary keyword, SQLAlchemy, 294
select method, SQLObject, 273
selectBy method, SQLObject, 274
self
  mocking calls to self, pMock, 196
semantic verifiers, 256
sendchange command, Buildbot, 114, 120
  paths used on build slave, 125
servers
  build servers, 103
  startup script, 112
sessionmaker function, SQLAlchemy, 285
sessions, SQLAlchemy, 285–286
setProperty method, BuildStep class, 361
setter mocking, PyMock, 229
setUp function, JavaScript, 330
setUp method
  correspondence of unittest/JsUnit, 326
  running unit tests manually in Eclipse, 155
  using PyMock with unittest, 230
setup.cfg file
  fixing options with setup.cfg, 97
  Subversion ignoring files, 98
setup.py program, Setuptools, 87
  bootstrapping Setuptools, 97
  building packages, 83
  building projects, 88–91
  console_scripts key, 91
  entry_points attribute, 91
  external_requirements attribute, 92
  gui_scripts key, 91
  installing executables, 91
  install_requires attribute, 92
  name attribute, 87
  version attribute, 87
Setuptools, 81
  basic usage, 87
  bdist_egg command, 90
  build command, 89
  building projects with setup.py, 89
  build_ext module, 89
  build_py command, 89, 90
  configuring values for options, 97
  description, 84
  development mode, 100–102
  dist directory, 91
  Distutils compared, 102
  easy_install program, 87
  eggs, 81
  egg_info command, 90
  finding packages on the Net, 94
  find_packages function, 88
  hook to application class, 175
  install command, 90
  installing, 86–87
  installing eggs, 84
  installing executables using setup.py, 91
  install_lib command, 90
  managing dependencies, 92–94
  packages directive, 88
  package_dir directive, 88
  removing existing package, 95
  running commands, 89
  running full test suite in development, 167
  setup.py program, 87
  Subversion ignoring files, 98–100
  version numbers, 88
  zip_safe flag not set warning, 94
set_count playback count, PyMock, 229
SGML (Standard Generalized Markup Language)
  HTML and, 310
  XML and, 311
shared projects, 59
ShellCommand build
  Buildbot running full test suite, 171
ShellCommand classes, 361
short iterations, 13–15
"Show all" errors window, JavaScript, 332
simple design, 12
site package mechanism, Python, 86
site-packages directory, Python, 82
  installing Python packages, 83
  installing rsreader into site-packages, 89
  replicable builds, 85
site-packages directory, Setuptools
  development mode, 100
  removing existing package, 95

slave *see* build slave, Buildbot
slave-lnx01
  aliasing hosts, 104
  configuring build slave, Buildbot, 112
  configuring build system, 106
slavePortnum property, Buildbot, 108, 109
slaves property, Buildbot
  configuring build master, 108
  configuring build slave, 112
socket keyword, DBMigrate, 301
software quality, 234
  measurements, 235
source
  aliasing hosts, 104
  naming Subversion for use with Buildbot, 104
source code
  build slave obtaining, 119
source control
  build master/slave accessing Subversion, 116–118
source folders, Eclipse, 33
source repository, 40
SQL injection attacks, 290
SQLAlchemy, 283–296
  all method, 288
  backref keyword, 294
  cascade keyword, 295
  chained expressions, 290
  connecting to database, 283
  delete method, 295
  filter method, 289
  filter_by method, 290
  first method, 288
  from_statement method, 290
  further documentation on, 296
  installing, 283
  join method, 295
  keyword queries, 290
  many-to-many relationships, 293
  mapper directive, 292
  MetaData class, 284
  nullable keyword, 285
  one method, 288
  one-to-many relationships, 291
  primary keys, 284
  query method, 287
    chained expressions, 290
  querying relations, 295
  results, 287
  secondary keyword, 294
  sessionmaker function, 285
  sessions, 285–286
SQLBuilder
  retrieving objects, SQLObject, 273
SQLExplorer, Eclipse plug-ins, 22
SQLite connection, setting up, 269
SQLObject, 267–282
  attribute defaults, 272
  connecting to database, 268, 269
  creating rows, 269–272
  deleting rows, 275
  describing database table to, 268
  foreign keys, 275
  installing, 267
  join statements, 279
  many-to-many relationships, 277
  multiple joins, 276
  multiple relationships, 280
  one-to-many relationships, 275
  retrieving objects, 273
  schema definition, 272
  updating fields, 274
src directory, Eclipse, 39
SSH trust relationships, 118
Standard Input and Output field
  Common tab, External Tools dialog, 170
start-commit hook, Buildbot, 121
startup screen, Eclipse, 24
startup script, servers, 112
start_response() callback, 315
state
  HTTP protocol, 312
statement coverage, 237
status command, Subversion, 49, 50, 53, 54, 56
  ignoring files, 98
status property, Buildbot, 110
stderr
  Nose intercepting, 160
stdin, HTTP request, 313
stdout
  defining print_entry_listings, pMock, 208
  HTTP response, 313
  Nose intercepting, 160

running unit tests manually in Eclipse, 153, 158
stdout attribute, pMock, 209
steps, Buildbot *see* build steps, Buildbot
stories, 339
stress testing, 140
string expansion
creating migrations, DBMigrate, 302
StringIO object
running unit tests manually in Eclipse, 153
stubs, 192
subclassing, 193
Subversion, 41
accessing repositories, 45
add command, 50
benefits of, 42–44, 79
checkout command, 47, 61
commit command, 50, 121
copy command, 51, 78
creating repository, 44
delete command, 52, 77
diff command, 54–56
directory structure, 44
events and hooks, 121
ignore property, 99
ignoring files, 98–100
import command, 46
importing into Eclipse from, 60–64
info command, 49
installing, 44–47
list command, 46
log command, 56
merge command, 56
mkdir command, 46
move command, 51
organizing projects, 45
parts of Subversion URL, 45
propset command, 99
resolved command, 58
revert command, 53
sharing subverted project, 59–60
status command, 49, 50, 53, 54, 56
svn directories, 43
update command, 54, 56
validating committed code, 256–257
working from command line, 47–59
adding files, 50–51
conflicting file changes, 55–59
copying/moving files, 51–52
deleting files, 52
examining workspace files/directories, 49–50
merging code, 55–59
modifying files, 53–54
restoring files, 53
updating working copy, 54–55
working through Eclipse, 64–79
adding files, 68–69
committing changes, 70–71
copying files, 78–79
deleting files, 76–77
editing files, 71–72
moving files, 77
renaming files, 77
resolving conflicts, 73–76
reverting changes, 72–73
reverting moves/renames/copies, 79
team repository view, 65–68
Subversion clients
accessing local filesystem, 117
accessing svnserve, 117
Subversion group
adding committers to, 117
permissions, 116
Subversion repository
access methods, 116
authorization/permissions, 116
baseURL property, 119
build master/slave accessing, 116–118
changing permission ownership, 117
defaultBranch property, 119
naming for use with Buildbot, 104
repository tree, 116
Subversive plug-in, 22
Subversion revision control systems, 41
team providers, 59
suite function, JavaScript
aggregating JsUnit tests, 335, 336
SVN build step
paths used on build slave, 124
publishing reports, 361
using source code, 119
svn directories, 43

svn group
  Subversion repository, 116
  switch command, 118
SVN step, Buildbot, 119–121
svn user, creating, 117
Svnhooks, 256
svnlook
  precommit hooks, 256
svnserve
  build master/slave accessing Subversion, 116
  committers and file ownership, 116
  daemon option, 117
  starting after host machine reboot, 118
  Subversion permissions, 116
switch command, svn, 118
Synchronize view
  working with Subversion through Eclipse, 65, 68
sys.argv file, 154
sys.stdout file, 153, 158
system metaphor, 8

T

Table method, SQLAlchemy, 284
tables
  creating tables, SQLObject, 270
  many-to-many relationships, SQLAlchemy, 293
  many-to-many relationships, SQLObject, 277
  multiple relationships, SQLObject, 280
  one-to-many relationships, SQLAlchemy, 291
  one-to-many relationships, SQLObject, 275
tags, HTML, 310
tags, Nose, 160
  tag expressions, 162
tasks
  job management system, 22
TDD (test-driven development), 10–11, 146–147
  description, 173
  implicit goal, 218
  using mock objects
    pMock, 195–212
    PyMock, 212–228
team providers, 59
team repository view, 65–68
tearDown method
  correspondence of unittest/JsUnit, 326
  running unit tests manually in Eclipse, 155
  using PyMock with unittest, 230
technical authority, 245
templates, Pydev, 254
test assertion
  implementing application tests, 176
test coverage *see* code coverage
test doubles *see* impostors
test fixtures, 142
test from_urls test, pMock, 204
test harness
  converting URLs into feed objects, 185
  Setuptools, 81, 102
test methods/functions
  correspondence of unittest/JsUnit, 326
test package
  running unit tests manually in Eclipse, 152
test runners
  Framework for Integrated Tests (FIT), 341
  JsUnit, 322
test suite pages
  aggregating JsUnit tests, 335–336
test-driven development *see* TDD
TestCase classes
  Nose, 149, 159
  unittest, 148
testGeneratesWithRaisedTermination, PyMock, 230
testing
  acceptance testing, 140, 339–353
  application tests, creating, 175
  customer tests, 139
  databases, 264
  exploratory testing, 140
  Framework for Integrated Tests (FIT), 340–353, 367
  functional testing, 139, 339
  integration testing, 140, 339
  isolating components, 190–192
  JavaScript, 320–326
  load testing, 140
  pending prefix, 195
  performance testing, 140, 339

programmer tests, 139
PyFit, 340–353
refactoring/condensing tests, pMock, 206
regression testing, 10, 140
running tests by URL, 336
running tests manually in Eclipse, 151–159
stories, 339
stress testing, 140
unit testing, 9–10, 141–142
using data files, 189
web applications, 316
graphics and images, 317
markup, 317
MVC, 316
XUnit test pattern, 332
testPage parameter, JsUnit
running tests by URL, 337
TestRunner objects, unittest, 148
TestSuite classes, unittest, 148
tests_require property, Nose, 167
test_add, pMock, 201
test_add, PyMock, 221
test_add_single_feed, pMock, 199
test_add_single_feed, PyMock, 217
test_aggregate_feed_creates_factory, pMock, 206
test_aggregate_feed_initializes_feed_parser, pMock, 203, 206
test_combine_feeds, pMock, 196
test_create_entry, pMock, 200
test_entries_is_always_defined, pMock, 201
test_entries_is_always_defined, PyMock, 221
test_feed_entry_constructor, PyMock, 222–223
test_feed_entry_from_parsed_feed, pMock, 208
test_feed_entry_listing, pMock, 202
test_feed_entry_listing, PyMock, 222
test_feed_listing, 181–182
test_feed_writer_intializes_stdout, pMock, 209
test_feed_writer_prints_nothing_with_an_empty_feed, pMock, 209
test_from_urls, PyMock, 216, 219
test_getting_attributes, PyMock, 229
test_get_feeds_from_urls, pMock, 203
test_is_empty, pMock, 210
test_is_empty, PyMock, 226
test_listing_for_item function, 182
test_listing_from_item function, 183
test_main, pMock, 212
test_main, PyMock, 227
test_new_main, pMock, 210
test_new_main, PyMock, 226
test_print_agg_feed_listing_is_printed, PyMock, 224
test_print_entry_listings, pMock, 208, 210
test_rsreader_dependency_initialization, PyMock, 227
test_rsreader_initializes_dependencies, pMock, 211
test_setting_attributes, PyMock, 229
test_suite property, Nose, 167
triangulation, 182
Twisted, 104, 364
typing
duck typing, 193
typographical naming conventions, Python, 248

U

umask command
Subversion group permissions, 116
undefined value, JavaScript, 327
underscore (_) prefix, Nose, 195
unit of work pattern, ORMs, 266
unit tests, 9, 141–142
arguments against unit testing, 143–146
don't understand code behavior, 144
environment for running tests, 144
not developer's job, 144
time spent on unit testing, 143, 144
unreliable tests, 144
asserting success or failure, 147
cohesion and coupling, 141
collective code ownership, 143
description, 139, 173
disabling unit tests, 145
discovering unit tests, 149
extent, 145
feedback on defective code, 251
finding tests with Nose, 159–160
fixing broken tests, 145
how they work, 141
isolating components, 190–192

JavaScript, 337
locating test code, 148
observing failing test first, 152
preventing regression, 145
problems with not unit testing, 142–143
Python packages for, 148
renaming values for readability, 183
RSReader with unit test skeleton, 152
running DBMigrate with unit tests, 305
running full test suite at build time, 171–173
running full test suite in development, 167–171
running tests after every change, 163–166
running tests by URL, 336
running tests manually in Eclipse, 151–159
size of unit test suite, 353
structure, 174
test-driven development, 146–147
test fixtures, 142
testing JavaScript, 320–326
tools producing unit tests from finished code, 145
web application frameworks, 337
when unit tests need to be run, 162
why unit testing fails, 144
writing, 10

unittest
correspondence with JsUnit, 326
description, 148, 174
introduction, 148
TestCase classes, 148
TestRunner objects, 148
TestSuite classes, 148
using PyMock with unittest, 230

unzip command
publishing reports, 360

update command, Subversion, 54, 56
updating fields, SQLObject, 274
upload directories
retrieving reports, 360

URIs (Universal Resource Identifiers)
options query, 311
schemes identifying resource type, 311
secondary index, 311
syntax and URLs, 311
Web and, 309, 337
when URI becomes URL, 312

URLs (Universal Resource Locators)
converting URLs into feed objects, 185, 188
feeds_from_urls method, pMock, 203
from_urls method, PyMock, 216
parts of Subversion URL, 45
reimplementing from_urls method, pMock, 204
running tests by URL, 336
steps followed by Web browsers, 310
test from_urls, pMock, 204
URI syntax, 311
Web and, 309, 337
when URI becomes URL, 312

user keyword, DBMigrate, 301
user stories, 7–8
usermod command, Linux, 117
use_pymock decorator, PyMock, 212

V

validateSlope function, JavaScript, 331, 335
validating committed code, 256–257
validation function, JavaScript, 329
velocity
development velocity, 242–243

verbose flag, DBMigrate, 304
verify method, pMock, 196
verify method, PyMock, 213
version attribute, setup.py, 87
version control
see also revision control systems
benefits of, 41

versions
egg naming structure, 84
replicable builds, 85–86
version numbers, 88

versiontable keyword, DBMigrate, 301
views, Eclipse, 24
virtual Python solution
replicable builds, 85

vocabulary
writing HTML spec document, 342

W

waterfall display, Buildbot, 111
description, 110
enhancing step progress description, 134

illustration of, 111, 114, 119, 122, 125, 128, 136
running FIT with Buildbot, 355
waterfall methodology, 2
Web, 309, 337
web application frameworks, 313, 337
web applications, 312–314
testing, 316–320
graphics and images, 317
markup, 317
MVC, 316
using write() callback, 315
variations on implementation, 312
Web Server Gateway Interface (WSGI), 314–315
WSGI middleware, 316
web browsers
description, 310
JavaScript and, 312
JavaScript interaction with, 328
steps followed by, 310
Web Server Gateway Interface *see* WSGI
web servers
application framework connections, 313
using write() callback, 315
web applications and, 312
WebStatus class
configuring build master, Buildbot, 110
will clause, pMock, 196
Wing IDE, 22
with-coverage option, nosetests, 258
WithProperties class
publishing reports, 361
with_sqlobject method
creating rows, SQLObject, 271
schema definition, SQLObject, 272
workbench, Eclipse, 25
working copy
updating working copy of file, 54–55
Working Directory field
builder properties window, 165
External Tools dialog, Eclipse, 169
World Wide Web, 309, 337
write() callback, 315
WSGI (Web Server Gateway Interface), 314–315
pronunciation of WSGI, 314
web application frameworks, 337
WSGI middleware, 316

X

xkcd_output
renaming printed_items value, 183
XML (eXtensible Markup Language), 311
SGML and, 310
XPath, 319
XUnit test pattern, 332

Z

zero_or_more playback count, PyMock, 229
zip_safe flag not set warning, Setuptools, 94
Zope Interface, 104